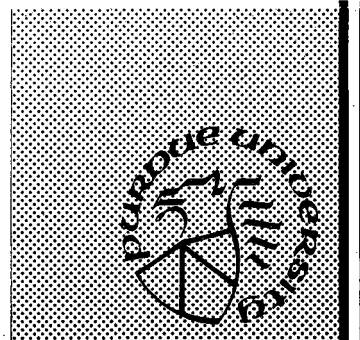
10-1-1987

Preliminary Report on High-Performance Computational Structures for Robot Control

Mahibur Rahman
Purdue University

David G. Meyer
Purdue University



Preliminary Report on High-Performance Computational Structures for Robot Control

Mahibur Rahman
David G. Meyer

TR-EE 87-38
October 1987

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABSTRACT

In this report we present some initial results of our work completed thus far on "Computational Structures for Robot Control". A SIMD architecture with the crossbar interprocessor network which achieves the parallel processing execution time lower bound of $o(\lceil a_1 n \rceil)$, where a_1 is a constant and n is the number of manipulator joints,for the computation of the inverse dynamics problem, is discussed. A novel SIMD task scheduling algorithm that optimizes the parallel processing performance on the indicated architecture is also delineated. Simulations performed on this architecture show speedup factor of 3.4 over previous related work completed for the evaluation of the specified problem, is achieved. Parallel processing of PUMA forward and inverse kinematics solutions is next investigated using a particular scheduling algorithm. In addition, a custom bit-serial array architecture is designed for the computation of the inverse dynamics problem within the bit-serial execution time lower bound of $o(c_1 k + c_2 kn)$, where c_1 and c_2 are specified constants, k is the word length, and n is the number of manipulator joints. Finally, mapping of the Newton-Euler equations onto a fixed systolic array is investigated. A balanced architecture for the inverse dynamics problem which achieves the systolic execution time lower bound for the specified problem is depicted. Please note again that these results are only preliminary and improvements to our algorithms and architectures are currently still being made.

CHAPTER 1
INTRODUCTION AND BACKGROUND

1.1 Introduction

Inverse Dynamics, Forward and Inverse Kinematics are particular robot control algorithms which need to be computed during regular robot servo control loops. Unfortunately, the computational complexity of these algorithms (see Table 1.1) degrades the servo control loop time of present-day robot control systems. This preliminary report therefore investigates high-performance computational structures to evaluate the specified algorithms in minimal time. The architecture, operation, performance, and derivation of lower bounds for the execution times and number of processors on specific computational structures to evaluate the specified problems is discussed.

The forward and inverse kinematics algorithms deal with the analytical study of the geometry of motion of a robot with respect to a fixed reference coordinate system. The former algorithm determines the position and orientation of the end-effector of the manipulator with respect to a fixed reference coordinate system, given the joint angles and geometric link parameters. Conversely, the latter algorithmic formulation determines the joint angles, given the position and orientation of the manipulator with respect to a fixed coordinate system. The second scheme is thus used primarily to determine whether the manipulator can reach the desired hand position and orientation, and if it can, find how many different manipulator configurations the given manipulator position and orientation may satisfy. Since the independent variables in a robot arm are the joint variables, and a task is usually stated in terms of the reference coordinate system, the inverse kinematics problem is used more frequently.

Table 1.1 Computational Compexity of Robot Control Algorithms

Operation	Inverse Dynamics (Newton-Euler)	PUMA Forward Kinematics	PUMA Inverse Kinematics
Type of Equations	Linear Recurrence	Non-Recursive Non-Linear	Non-Recursive Non-Linear
Multiplications	678	100	74
Additions	597	77	66
Transcendental	–	137	62
Square Root	–	–	3
Division	–	–	10

The inverse kinematics solution may be obtained by various methods such as inverse transform, screw algebra, dual matrices, dual quaternian, iterative, and geometric approach. The iterative solution often requires a very high number of computations and does not guarantee convergence to the correct solution. The inverse transform method yields a set of explicit, non-iterative joint angle equations. The dataflow structure of this technique will therefore be used in this thesis. The architecture, operation, and performance of a parallel processing architecture to compute the specified kinematics algorithms for the PUMA robot is described.

The inverse dynamics problem is that of efficiently determining the motor torques required to drive a manipulator arm in free motion. These torques must be evaluated repeatedly during servo control loops to avoid undesirable motion deviations of the robot arm from the desired trajectory path. The computation of the torques is, however, a very mathematically intense task which degrades the servo loop time of present-day robot control systems. Many researchers have thus concentrated on simplifying the dynamics equations [30,35,57], or developing new computational architectures [5,19,26,32,34,35,43,45,58,60] to reduce the servo sampling period. In this paper, the latter approach is employed.

There are a number of ways to formulate the robot arm dynamics equations of motion. They include the Lagrange-Euler [4], recursive Langrange-Euler [18], Newton-Euler [36] and the generalized d'Alembert principle functions [30]. Among these methods, the Newton-Euler (N-E) dynamics equations is the most efficient and has been shown to possess the computational time lower bound of O(n) [36], where n is the number of degrees-of-freedom of the manipulator. The N-E algorithm will therefore be used to implement the various computational structures discussed in this thesis. It should, however, be noted that the proposed architectures may be customized to meet the computational model of other formulations of the inverse dynamics problem.

The N-E formulation uses two types of iterative recursions, namely, the forward and the backward recursions, which are applied to the robot links sequentially. The forward recursion propagates kinematics information — such as linear velocities, angular accelerations, and linear accelerations at the center of mass of each link — from the inertial coordinate frame to the hand coordinate frame. The backward recursion propagates the forces and moments exerted on each link from the end-effector of the manipulator to the base reference frame. The parallel, pipelined, and recursive nature of the N-

E dynamics algorithm suggests that it is amenable to parallel processing structures using off-the-shelf processors, as well as parallel and pipelined custom VLSI array architectures. This thesis therefore also presents such computational schemes.

1.2 Previous Work

Several parallel architectures have been proposed to solve the inverse dynamics problem. Luh and Lin [35] proposed a modification of the traditional "branch and bound" scheduling technique to process the equations in parallel on a set of n CPUs. Kasahara and Narita [19] continued along the same path by proposing to use the DFIHS (Depth-First-Implicit-Heuristic-Search) scheduling algorithm for this purpose. However, in both the above studies, the important problem of interprocessor communication which degrades the performance of their particular multiprocessor structure is ignored.

Orin, Chao and Schrader [45], recognizing the pipelined nature of the N-E formulation, proposed assigning two pipelined CPUs per link to compute the forward and backward recursions in each processor, respectively. Their structure eliminates some of the performance degradation problems associated with interprocessor communications that exist in the computation of the N-E algorithm for parallel processing environments. The task of computing forward or backward iterations of a single stage of the N-E recursive algorithm at a single CPU location does not, however, produce sufficient speedup over a single CPU solution.

Liao and Chern [32] suggest using the CBAP (Cross-Bus Array Processor), which uses a large set of bit-parallel processors arranged in an array format with two sets of busses crossing over the array in two directions. The primary disadvantage of this system is the cost-inefficiency of using a large number of bit-parallel array processors which are not fully utilized. Additional problems include complexity in the operand data alignment process and the directional data-shifting mechanisms used in the control of the array, causing the system to be susceptible to hardware/software faults.

Lathrop [26] proposed two parallel algorithms on the inverse dynamics problem using a group of special purpose processors. One is a linear-parallel algorithm and the other is a logrithmic-parallel algorithm based on the

partial sum technique. The main concern with both approaches is the massive buffering between forward and backward recursions, which deteriorates the performance. The second problem is that they both involve complex interprocessor communication structures which frequently cause data to be fetched and, as a result, data for operand pairs are not properly aligned for parallel computations.

Lee and Chang [29] recently proposed using the recursive doubling algorithm with a modified inverse perfect shuffle interconnection scheme between a set of parallel processors. Their processor interconnection structure improves pipelining and eliminates some of the problems associated with interprocessor communication. They did not, however, incorporate any operand access mechanisms into their structure to allow proper synchronization of the parallel processors. Their implementation requires a total of 530 modular processors, where the processor complexity required to perform 3-D parallel vector dot product and/or vector addition along with the expensive interconnection structure among processors indicate this approach may not be a cost-efficient and fault-tolerant practical approach.

For the computation of the inverse kinematics algorithm, which is an equally important problem, Lee and Chang have proposed a pipelined VLSI architecture using CORDIC processors as building blocks of the structure. Delay buffers were inserted along data paths of the system to balance operand arrival time of the CORDIC computational nodes. The author has no valid criticism of this custom VLSI architecture.

1.3 Organization of Preliminary Report

Parallel processing using off-the-shelf processing elements to evaluate the inverse dynamics, PUMA forward and inverse kinematics algorithms is discussed in chapter 2. Ideal lower bounds on the number of parallel processors and execution time for the computation of the inverse dynamics problem in the specified computational model is derived here. A novel scheduling algorithm for the parallel processing of N-E equations on an SIMD machine with a particular interprocessor communication network is presented next. Simulation results performed on this architecture are then delineated and compared to the performance of previous related work completed on this problem. The second part of this chapter describes the performance of both types of kinematics algorithms in a multiprocessor environment with a

shared memory interprocessor communication mechanism while using a particular instruction scheduling algorithm.

In chapter 3, a custom bit-serial array architecture for the computation of the inverse dynamics problem is presented. The organization, operation and performance of the proposed system is described. In addition, the architecture and intercell communication protocol of an individual bit-serial cell (which is used as the building block of the overall array structure) is delineated.

Mapping of the Newton-Euler equations of motion onto a fixed systolic architecture is depicted in Chapter 4. The systolic design methodology for this mapping process is discussed here. This design procedure is used to find a basic set of systolic processor architectures which are used to build the complete systolic system. Integer linear programming is applied for the optimal buffer assignment problem to obtain a "balanced" systolic array for the computation of the specified problem. The performance, operation, design and lower bound task latency costs of such a systolic system are also delineated here. Finally, Chapter 5 provides a summary of conclusions drawn from this report.

CHAPTER 2
PARALLEL PROCESSING OF ROBOT INVERSE DYNAMICS, FORWARD AND INVERSE KINEMATICS COMPUTATIONS

2.1 Introduction

In the first part of this chapter, we discuss parallel processing of the inverse dynamics computational problem. First, the ideal lower bounds for the number of parallel processors and execution time to evaluate the specified problem in a parallel processing environment using an optimal scheduling algorithm is derived. The proposed multiprocessor architectural model is then presented. Next, a scheduling algorithm customized for optimal parallel processing on the proposed system model is described. Further, simulation results for computing the desired algorithm using the proposed architecture and scheduling methodology is delineated and compared to previous work completed on this topic.

In the second part of the chapter, the parallel processing system model for the computation of PUMA forward and inverse kinematics algorithms is presented. This architecural model for the evaluation of the specified problems is simulated by using the DFIHS (Depth-First-Implicit-Heuristic-Search) algorithm [19] as the primary task scheduling tool.

2.2 Parallel Processing of the Inverse Dynamics Problem

2.2.1 Ideal Lower Bounds on the Number of Processors and Execution Time of the Inverse Dynamics Computational Problem while running on a Multiprocessor System using an Optimal Scheduling Algorithm.

The ideal minimum number of processors required to compute the inverse dynamics problem in minimal time while running in a parallel processing environment using an optimal scheduling algorithm is discussed here. In addition, the limitation on speeding up the specified problem in the indicated computational architecture is also investigated. Before deriving these ideal lower bounds, the following notation, definitions, and lemmas must be established.

Notation:

m_p = Minimum number of parallel processors needed to compute the inverse dynamics problem in minimal time using an optimal scheduling algorithm.

T_p = Lower bound on the execution time of the inverse dynamics problem while running in a parallel processing system with m_p processors, using an optimal scheduling algorithm.

E<*l*> denotes a linear arithmetic expression of *l* distinct atoms, where an atom is a constant or variable, e.g. E<4> = a + b – c/d.

T_{cp} = Minimum time to perform the set of computations in the critical path of a task graph G.

Definition 2.1: The *activity of vertex j* [10] in a task graph is defined as,

$$f(\tau_j,t) = \begin{cases} 1 , & \text{for } t \in [\tau_j - t_j, \tau_j] \\ 0 , & \text{otherwise.} \end{cases}$$

where $f(\tau_j,t)$ indicates the activity (or computational delay) of vertex j (or task T_j) along time, according to the restrictions imposed by the task graph, and τ_j is the earliest completion time of task T_j.

Definition 2.2: The *load density function* [10] is defined by

$$F(\tau,t) = \sum_{j=1}^{l} f(\tau_j,t) \tag{2.1}$$

where $F(\tau,t)$ depicts the total activity of the task graph as a function of time, and l is the total number of tasks in task graph G.

Definition 2.3: The load density function within the interval $[\theta_1, \theta_2] \subset [0, t_{CP}]$ after all tasks have been shifted to yield minimum overlap with this interval [10] is defined as,

$$R(\theta_1, \theta_2, t) = \int_{\theta_1}^{\theta_2} F(\tau,t)\, dt \tag{2.2}$$

Lemma 2.1: A lower bound on the minimum number of processors [10] to execute a task graph G within time t_{CP} is given by,

$$m_p = \left\lceil \max_{[\theta_1, \theta_2]} \left[\frac{1}{\theta_2 - \theta_1} \int_{\theta_1}^{\theta_2} R(\theta_1, \theta_2, t)\, dt \right] \right\rceil$$

where the maximum is taken over all integer time segments $[\theta_1, \theta_2]$.

Theorem 2.1: The minimum number of parallel processors needed to compute the inverse dynamics problem within time t_{CP} of the Newton-Euler task graph using an optimal scheduling algorithm is n, where n is the number of manipulator joints.

Proof: Using Eq. 2.1, the load density function for the inverse dynamics problem is,

$$F(\tau,t) = \sum_{j=1}^{b_1 n} f(\tau_j,t) = b_1 b_2 n \tag{2.3}$$

where b_1 is the number of tasks in a single iteration of the N-E algorithm, and b_2 is the number of tasks in the complete N-E task graph which have an unit activity at a set vertices T_i. The second load density function within

time interval $[\theta_1, \theta_2] \subset [0, t_{CP}]$ to yield minimum overlap for the inverse dynamics problem is found by Eq. 2.2 to be,

$$R(\theta_1, \theta_2, t) = b_1 b_2 n(\theta_2 - \theta_1)$$

Thus, by lemma 2.1, m_p must be of the form,

$$m_p = \left\lceil \max[\theta_1, \theta_2] \left[\frac{1}{\theta_2 - \theta_1} \int_{\theta_1}^{\theta_2} b_1 b_2 n(\theta_2 - \theta_1)\, dt \right] \right\rceil$$

$$= \left\lceil \max_{[\theta_1, \theta_2]} [b_1 b_2 (\theta_2 - \theta_1) n] \right\rceil \qquad (2.4)$$

It is evident from Eq. 2.4 that the minimum number of parallel processors to compute the inverse dynamics algorithm using an optimal scheduling algorithm is *n*. Note that this lower bound occurs when $\max_{[\theta_1, \theta_2]} \left\lceil b_1 b_2 (\theta_2 - \theta_1) \right\rceil$ is unity.

Lemma 2.2: The set of computations of a graph G cannot be completed with m processors in a time less than t_L [10],

$$t_L = t_{CP} + \max_{t_2 \le t_K \le t_{CP}} \left[-t_K + \frac{1}{m} \int_0^{t_K} F(\bar{\tau}, t)\, dt \right]$$

where t_K is a discrete point in time and $\bar{\tau}$ is the latest completion time of a specific task.

Lemma 2.3: The critical path time, t_{CP}, of the inverse dynamics task graph S is of the form $a_1 n$, where a_1 is a specified constant and n is the number of manipulator joints.

Proof: Let x_1 be the computational delay in the critical path of a single iteration of the N-E dynamics algorithm. Thus, the critical path time for n iterations is $t_{CP}\langle n \rangle = x_1 n$. The inverse dynamics problem may be considered as computing a set of joint torques which result in obtaining the joint torques. Each joint torque τ_i of joint i can be expressed as an arithmetic expression containing at last 3n atoms: n joint positions $\{q_i\}_{i=1}^{n}$, n joint velocities $\{\dot{q}_i\}$,

and n joint accelerations $\{\ddot{q}\}_{i=1}^{n}$, implying, $t_{CP}<3n> = 3x_1n$. The previous expression may be rewritten without changing the order of the lower bound as $t_{CP}[\tau_1,\ldots,\tau_n] = a_1n$, where a_1 is specified constant.

Theorem 2.2: The minimum time to compute the inverse dynamics problem in a parallel processing system with m_p processors using an optimal scheduling algorithm is bounded below by $o\lceil a_1n \rceil$, where n is the number of manipulator links, and a_1 is a specified constant.

Proof: By substituting $F(\tau,t)$ from Eq. 2.3, t_{CP} from Lemma 2.3, and m from Theorem 2.1 into the expression for t_L in Lemma 2.2, we obtain a formulation for T_P,

$$T_P = a_1n + \max_{t_1 \le t_K \le t_{CP}} \left[-t_K + \frac{1}{n} \int_0^{t_K} b_1b_2ndt \right]$$

$$= a_1n + t_K(b_1b_2 - 1) \qquad (2.5)$$

Since a_1, b_1, b_2 and t_K are specified constants, the lower bound for T_P, without changing the order of the lower bound of Eq. 2.5, may be expressed as,

$$T_P[E<3n>] \ge o(a_1n)$$

where E<3n> represents the computational complexity of evaluating n joint positions $\{q_i\}_{i=1}^{n}$ n joint velocities $\{\dot{q}_i\}_{i=1}^{n}$, and n joint accelerations $\{\ddot{q}_i\}_{i=1}^{n}$.

2.2.2 SIMD Multiprocessor Architectural Model

The N-E Equations of motion may be decomposed into a set of homogeneous linear unidirectional and recurrence tasks (see section 4.4). This suggests that the inverse dynamics problem is amenable to a SIMD multiprocessor-based system. The basic architectural model of the proposed architecture to compute the desired problem is presented in this section.

It has been stated that "the most critical system control mechanism in a multiprocessor-based computer structure are clearly those involved with interprocess

and interprocessor communication"[17]. The interprocessor communication network for the SIMD model is therefore a crucial choice. In Theorem 2.1, we proved the ideal lower bound on the number of parallel processors, m, to compute the inverse dynamics problem to be n, where n is the number of manipulator joints. For n>6, industrial robots have undesired redundant degrees of freedom. It is therefore safe to assume $n \leq 8$ for worst-case applications. Thus, for $n=m \leq 8$, a "complete" interprocessor communication strategy such as the crossbar network is desired for this purpose due to the commercial availability of 8×8 crossbar switches. In most image processing applications, crossbar interprocessor communication networks are generally not used for SIMD processing due to excessive network overhead costs since $m \gg 8$. This is however not the case for our application, as discussed above.

Fig. 2.1 illustrates the proposed SIMD system model. The configuration is structured with m synchronized processing elements with a crossbar interprocessor communication network. The microcode ROM (Read-Only-Memory) contains a static instruction schedule for optimal SIMD processing of N-E Equations of motion on the proposed system model. Each processing element PE_i has its own local memory PEM_i. Interprocessor communication data is thus directed to single or multiple destination PEM_i from a single source PE_i at the end of every SIMD instruction cycle. Memory conflicts, however, occur when multiple source PE_i attempt to send data to the same destination PEM_i through the crossbar network. In addition, an unsystematic SIMD instruction schedule in the microcode ROM may cause PE_is to be idle during SIMD processing cycles, thus causing a longer overall parallel execution time. It is therefore clear that an algorithm which optimally schedules SIMD tasks to the PEs, in addition to scheduling intermediate results through the crossbar network without resource conflicts, is desired. The next section presents such an algorithm.

2.2.3 Proposed Instruction Scheduling Algorithm for SIMD Architecture with Crossbar Interprocessor Communication Network

In this section, a novel algorithm for the scheduling of SIMD-mode processes and destination processor data through the crossbar interprocessor network is presented. It should be stressed here that the technique for scheduling of destination data embedded in the algorithm is valid only for the case of a crossbar interprocessor network. Before a formal presentation of this algorithm, the following notation must be established.

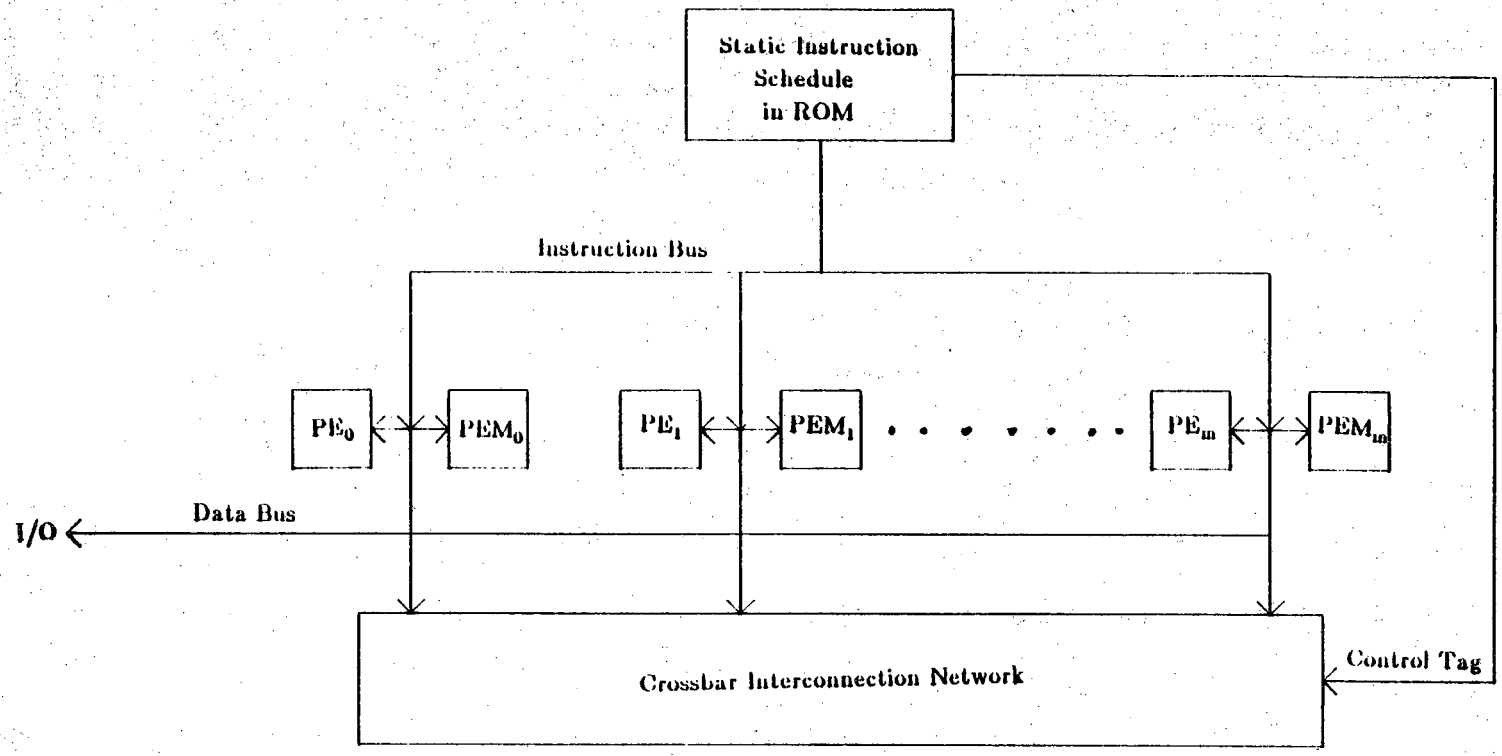


Fig. 2.1: SIMD Architectural Model for Robot Inverse Dynamics Computation

Notation:

(1) $T_R(t) = \{T_1,\ldots,T_l\}$, $l \in z^n$, is a set of ready tasks (tasks whose predecessors have been executed) at time t which perform the same type of operation.

(2) $T_{RF}(t) = \{T_{R1},\ldots,T_{Rd}\}$, $k \in z^n$, is a set of valid T_R's at time t.

(3) N_{Ri} specifies the number of ready tasks in the task set $T_{Ri} \in T_{RS}(t)$.

(4) $N_I(t + \Delta)$ denotes the number of interprocessor data transfers required to be performed by the crossbar network at time, $t + \Delta$, where Δ is the computational delay of the operation initiated at time t.

(5) $N_{Ci}(t + \Delta)$ is the number of unique destination processors to which the result of task T_i executed in processor P_j is to be sent at time $t + \Delta$.

(6) $P_{Di}(t + \Delta) = \{P_1,\ldots,P_r\}$, $r \in z^n$, specifies the set destination processors to which the result of task T_i executed in processor P_j is to be sent at time $t + \Delta$.

(7) WG is a weighted precedence graph for a computational problem C, where specifications of node task number, node computational delay and node task type must be included in the task graph.

Procedure ISSMACIN (Instruction Scheduling for a SIMD Multiprocessor Architecture with Crossbar Interconnection Network).

Input: Weighted Graph WG and number of available parallel processors m.

Output: Set of tables T = {STAT, DPAT} where,

STAT is a SIMD Task Allocation Table which specifies the assignment of SIMD processes to parallel processors at time t.

DPAT is a Destination Processor Assignment Table which specifies the destination processor tags for the proper routing of data through the crossbar network at time $t + \Delta$.

STEP 1: By the CP/MISF method [19], determine the ''level'' of each task in WG, where the level l_i of task T_i is defined as the longest path from the exist node to node N_i corresponding to T_i. Mathematically,

$$l_i = \max_k \sum_{j \in \pi_k} t_j$$

where π_k is the kth path from the exit node to node N_i, and t_j is the

computational delay of node j.

STEP 2: Construct the priority list L in the descending order of l_i and number of immediately successive tasks n_i. Order tasks of identical l_i and n_i in lexicographic fashion.

STEP 3: Initial assignment process of tasks in WG to parallel processors for SIMD program execution:
REPEAT

(a) Determine $T_{RS}(t) = \{T_{R1},\ldots,T_{Rk}\}$

(b) Find $T_{Ri} \in T_{RS}(t)$ with $\max[N_{Ri}]$.

(c) Select max[x] tasks from T_{Ri} where $x \in z^n$ and $1 \le x \le m$. Let T_x be the chosen.

(d) Assign the tasks T_x to x parallel processors in lexicographic order.

(e) Direct the corresponding SIMD command to these x processors and mask the other m-x processing elements.

(f) Perform $T_{Ri} = T_{Ri} \backslash T_x$.

UNTIL $N_{Ri} = 0$ for all possible $T_{Ri} \in T_{RS}$ and t.

STEP 4: Rearrange the task assignment of STEP 3 at each SIMD processing state t so that $N_I(t + \Delta)$ is maximized. This may be achieved by using the generalized backtracking algorithm [15]. The explanation of this algorithm is beyond the scope of this thesis. We may however note that the time complexity of the specified algorithm is $O(m \cdot s)$, where m is the number of PEs, and s is the number of SIMD processing states t.

STEP 5: The scheduling of destination processor data through the crossbar interconnection network at time $t + \Delta$ is as follows:

REPEAT

(a) Find task $T_i \in T_x(t + \Delta)$ with $\max[N_{Ci}]$ where T_x is the set of x tasks computed at time t.

(b) Broadcast result of task T_i executed in processor P_l, $1 \le l \le m$, to the set of destination processors $P_{Di}(t + \Delta)$.

(c) At the same time, send/broadcast results of set of tasks T_j, where $T_j \subset T_x$ and $T_i \notin T_j$, to set of destination processors P_{Dj}, where $P_{Dj} \subset P_{Di}(t + \Delta)$.

(d) $T_x(t+\Delta) = T_x(t+\Delta) \backslash (T_i \cup T_j)$.

UNTIL $N_{Ci} = 0$ for all possible $T_i \in T_x$, t and Δ.

Let us now illustrate procedure ISSMACIN by a simple example. Fig. 2.2 shows a given WG. The task numbers T_i are specified within the nodes N_i. The nomenclature t_y *j* next to each node specifies the type of task to be computed in the vertex, where *j* denotes the type of task. For purpose of simplicity, we shall assume the operational delay of task type *j* is *j* time units. Applying ISSMACIN to Fig. 2.2 when two parallel processors with a crossbar interprocessor communication network is presumed to be available:

STEP 1: Applying CP/MISF method to Fig. 2.2:
$l_1 = 5, l_2 = 6, l_3 = 5, l_4 = 5, l_5 = 4,$
$l_6 = 5, l_7 = 4, l_8 = 4, l_9 = 4, l_{10} = 3,$
$l_{11} = 2, l_{12} = 3, l_{13} = 3, l_{14} = 2, l_{15} = 2.$

STEP 2: The priority list L of the task set is:
L = {T2, T1, T3, T4, T5, T6, T7, T8, T9, T10, T12, T13, T11, T14, T15}

STEP 3: Results for the first-run through the loop which specifies the process for assigning tasks to processors, we get:

(a) $T_{RS}(t_1) = \{T_{R1}, T_{R2}\}$ where,
T_{R1} = {T1, T3, T4, T5} and
T_{R2} = {T2}.

(b) $N_{R1} = 4$, and $N_{R2} = 1$. Thus, T_{R1} is selected for SIMD processing.

(c) T_x = {T1, T3, T4}.

(d) Assign T1 to PE1, T3 to PE2, and T4 to PE3.

(e) Direct type 1 SIMD command to PE1-3.

(f) $T_{R1} = T_{R1} \backslash T_x$ = T5.

By repeating the loop, we may arrive at the initial task assignment table of Table 2.1 for SIMD processing states t1-t5. Note that the last column of the table specifies the type of SIMD task performed at state t*i*.

STEP 4: By apply the generalized backtracking algorithm to Table 2.1 to minimize the number of interprocessor data transfers at time $ti + \Delta$, we may arrive at the more efficient task assignment (STAT) of Table 2.2.

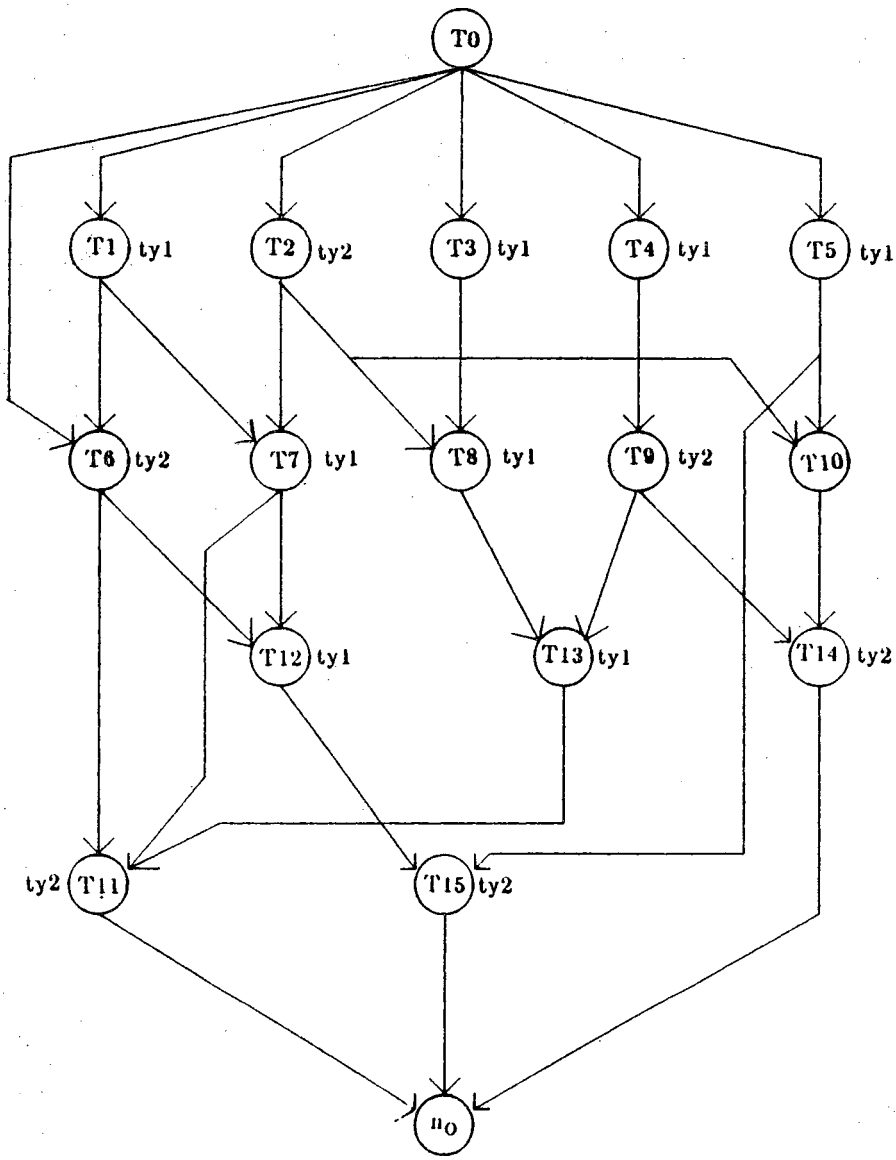


Fig. 2.2: A given WG

Table 2.1: Initial SIMD Task Assignment for WG of Fig. 2.1.

Time	PE 1	PE 2	PE 3	Task Type
t1	T1	T3	T4	ty1
t2	T2	T6	T9	ty2
t3	T5	T7	T8	ty1
t4	T10	T12	T13	ty1
t5	T11	T14	T15	ty2

Table 2.2: Final STAT for WG of Fig. 2.1.

Time	PE 1	PE 2	PE 3	Task Type
t1	T3	T1	T4	ty1
t2	T9	T6	T2	ty2
t3	T5	T7	T8	ty1
t4	T12	T13	T10	ty1
5	T11	T14	T15	TY2

Table 2.3: DPAT for WG of Fig. 2.1

Time	PE 1	PE 2	PE 3
t1+Δ1	d3	d2	d1
t2+Δ1		d1	d2,d3
t2+Δ2	d2		
t3+Δ1	d3	d1	d2
t4+Δ1	d3	d1	d2

Table 2.4: STAT for a Single Iteration of the N-E Dynamics Algorithm

Time	PE 1	PE 2	PE 3	PE 4	PE 5	PE 6	PE 7	PE 8	Task Type
t1	1	2	4	5	6	7	9	10	M
t2	3	8	20	23	25	27	28	I	A
t3	21	22	24	26	29	30	37	38	M
t4	39	40	41	42	11	12	13	14	M
t5	34	35	36	46	47	48	31	I	A
t6	49	50	51	52	53	54	55	56	M
t7	57	58	59	60	15	16	17	18	M
t8	70	71	72	73	74	75	76	77	A
t9	19	82	83	84	85	86	87	88	M
t10	89	90	91	92	93	94	95	96	M
t11	97	98	99	100	101	102	103	104	M
t12	105	61	62	63	64	65	66	67	M
t13	32	33	121	122	123	124	125	126	A
t14	127	128	129	43	44	45	79	80	A
t15	81	118	119	120	139	140	141	142	A
t16	109	110	111	112	113	114	68	69	M
t17	106	107	108	143	144	151	152	153	A
t18	130	131	132	133	134	135	115	116	M
t19	136	137	138	145	146	147	157	158	A
t20	148	149	150	159	178	179	180	I	A
t21	160	161	162	172	173	174	175	176	M
t22	154	155	156	184	185	186	I	I	A
t23	177	163	164	187	188	189	190	191	M
t24	206	196	197	198	199	200	201	193	A
t25	205	206	207	208	209	210	211	212	M
t26	192	165	166	213	167	168	169	170	M
t27	217	218	219	214	215	216	182	183	A
t28	220	221	222	165	166	I	I	I	A
t29	223	224	225	202	203	204	I	I	A
t30	226	227	228	I	I	I	I	I	A
t31	229	I	I	I	I	I	I	I	A
t32	230	I	I	I	I	I	I	I	A

STEP 5: Applying the loop at $t2 + \Delta 1$:

(a) $N_{C2} = 2$, $N_{C6} = 1$, $N_{C9} = 1$. Thus, task T2 is thus selected for primary data broadcasting.

(b) Results of T2 is therefore sent to processing elements 2 and 3.

(c) Since $P_{D6} \subset P_{D2}$ and $P_{D9} \subset P_{D2}$, product of T6 is thus also selected for sending its results to PE1 at this time.

(d) $T_x(t2 + \Delta 1) = T_x(t2 + \Delta 1) \backslash (T2 \cup T6) = T9$

At $t2 + \Delta 2$, result of T9 is naturally routed to PE2. By repeating the specified loop, we may arrive at the DPAT in Table 2.3. The nomenclature d*i* in the table columns specify the destination processor memories PEM_i to which respective source processor sends its result at time $tx + \Delta$.

When procedure ISSMACIN is applied to parallel process a single iteration of the N-E algorithm, the resulting STAT for this application is illustrated in Table 2.4. In this Table, the initials M and A in the "task type" column specifies a multiplication and addition SIMD operation respectively. Notice the high density of SIMD task processing when using the proposed scheduling algorithm to evaluate the 230 tasks in a single iteration of the N-E algorithm. The idle processors prevalent from t28 through t32 will not occur when the number of iterations is more than one (which is the case for any n-link manipulator), since tasks of successive iterations are computed in SIMD processing fashion at the indicated idle processors at specified time intervals. Accordingly, the DPAT generated by procedure ISSMACIN for a single iteration of the N-E algorithm is shown in Table 2.5. This table verifies that an average of 3 interprocessor data transfers through the crossbar network are required at the end of every SIMD instruction processing cycle. From these two tables, it may be deduced that the worst-case execution time to compute the inverse dynamics problem of an n-link manipulator on the proposed architecture using procedure ISSMACIN as primary design tool for SIMD processing is $(7t_a + 15t_m + 32t_f + 65t_s)n$, where, t_a is the time to perform an add operation, t_m is the time to compute a multiplication operation, t_f is the time to fetch operands from local PEM_i, and t_s is the time to send the result to the destination PEM_i(s) through the crossbar network.

2.2.4 Simulation Results and Comparison to Previous work

In this section, we present simulation results for computing the inverse dynamics problem using the proposed SIMD architectural model and procedure ISSMACIN.

Table 2.5: DPAT for a Single Iteration of the N-E Algorithm

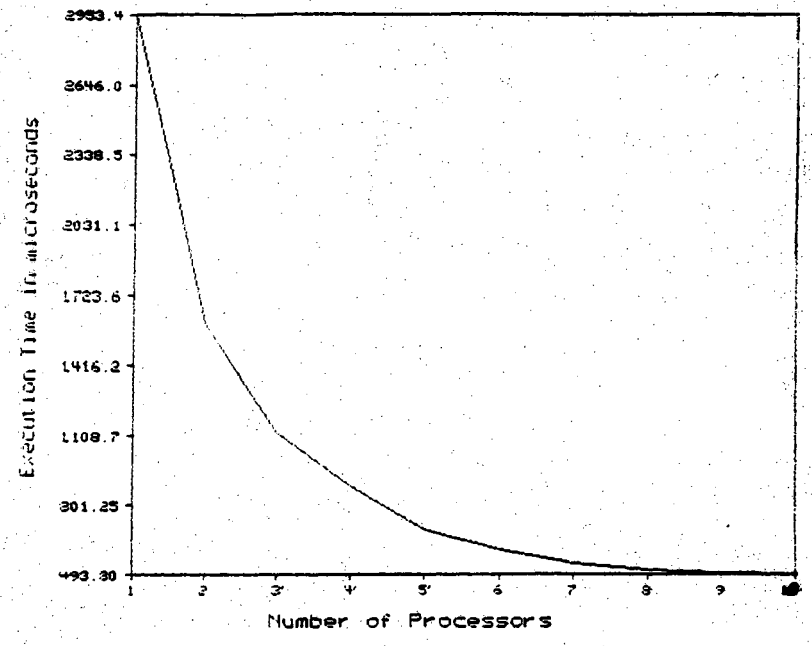
Time	PE1	PE2	PE3	PE4	PE5	PE6	PE7	PE8
t1+Δ1	d3		d4		d5		d6	d7
t1+Δ2		d3		d4		d5		
t2+Δ1	d1,d2,d3	d4,d5				d7	d8	
t2+Δ2			d1	d2	d3			
t3+Δ1	d1	d2	d3				d4	
t3+Δ2				d1	d2	d3		d4
t4+Δ1	d5		d6		d7		d4	d1
t4+Δ2		d5		d6		d7		
t5+Δ1		d6	d1-5,d7,d8					
t5+Δ2	d2-5,8	d1,d7						
t5+Δ3		d3,d4		d1	d2			
t5+Δ4						d3	d4	
t6+Δ1	d4		d5		d6		d7	
t6+Δ2		d4		d5		d6		d7
t7+Δ1	d8		d7		d1	d5	d2	
t7+Δ2		d8		d7				d2
t8+Δ1	d1,d4-8	d2	d3					
t8+Δ2		d1,d4,d5,d7,d8						d3
t8+Δ3			d4-6	d3	d8	d1,d2		
t8+Δ4				d4	d5		d1,d2	d6
t9+Δ1	d6	d2		d3		d4		d1
t9+Δ2			d2		d3		d4	
t10+Δ1	d3	d4		d5	d5	d6		d7
t10+Δ2			d4				d6	
t11+Δ1	d7	d8		d1		d2		d3
t11+Δ2			d8		d1		d2	
t12+Δ1	d3		d4	d1	d8		d2	
t12+Δ2		d3				d8		d1
t13+Δ1	d5	d6			d7	d8	d4	d3
t13+Δ2			d5	d6				
t14+Δ1	d8	d4	d5		d6	d7	d1	d2
t14+Δ2				d5				

Table 2.5: DPAT for a Single Iteration of the N-E Algorithm (continued)

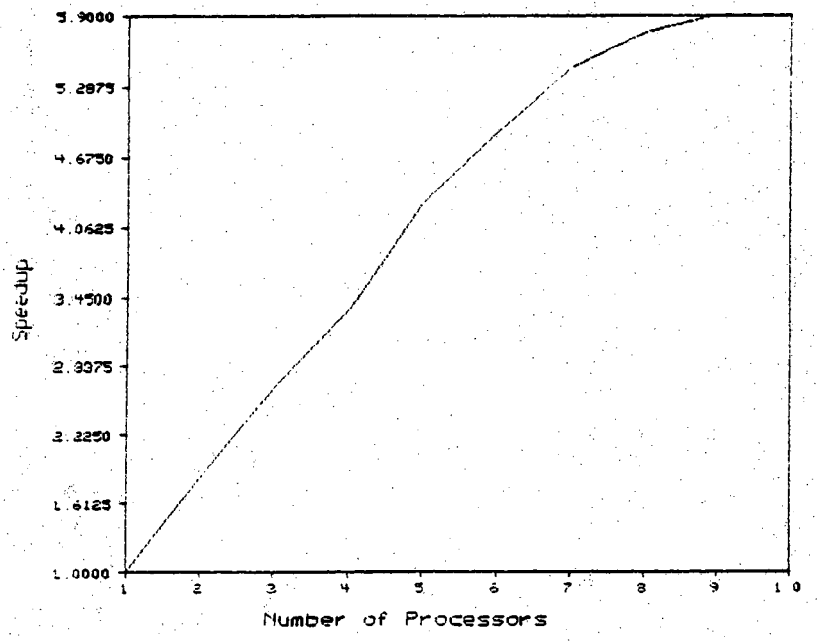
Time	PE1	PE2	PE3	PE4	PE5	PE6	PE7	PE8
t15+Δ1	d3	d6	d6	d8				
t15+Δ2					d6	d7	d8	
t15+Δ3								d7
t16+Δ1	d1			d2			d3	
t16+Δ2		d1			d2			d3
t16+Δ3			d1			d2		
t17+Δ1	d4,d5	d1	d2,d3	d8		d7		
t17+Δ2		d5			d4		d8	d4
t18+Δ1	d4		d5		d6		d3	
t18+Δ2		d4		d5		d6		d3
t19+Δ1	d1	d2	d3					
t19+Δ2				d1	d2	d3		
t19+Δ3							d1	d2
t20+Δ1	d1	d2	d3		d7,d8	d4	d5,d6	
t20+Δ2				d3		d1		
t21+Δ1	d4	d5	d6					
t22+Δ2				d4		d5		d6
t22+Δ3					d4		d5	
t23+Δ1	d3	d1		d5		d6		d7
t23+Δ2			d1		d5		d6	
t24+Δ1		d1,d4,d7	d2,d5,d8	d3,d6				
t24+Δ2	d8			d4	d2	d5	d6	d3
t25+Δ1	d1		d2		d3		d4	
t25+Δ2		d1		d2		d3		d4
t26+Δ1	d7		d4	d3	d7	d4	d8	
t26+Δ2		d7						d8
t27+Δ1	d1	d2	d3				d4	d5
t27+Δ2				d1	d2	d3		
t28+Δ1	d1	d2	d3	d5	d6			
t29+Δ1	d1	d2	d3					
t30+Δ1	d1							
t30+Δ2		d1						
t31+Δ1	d1							

Table 2.6: MC 68020 Operation Execution Times

Function	Assembler Code	Operational Delay (in # of Clock Cycles)	Operational Delay (in μs)
Fetch Operands from Shared M or PEM_i	MOVE <EA>,Dx MOVE <EA>,Dy	14	0.84
Store Result in Shared M or Destination PEM_i	MOVE Dy,<EA>	5	0.3
Add	ADD Dx,Dy	3	0.18
Multiply	MUL Dx,Dy	28	1.68



(a): Execution Time versus Number of Processors



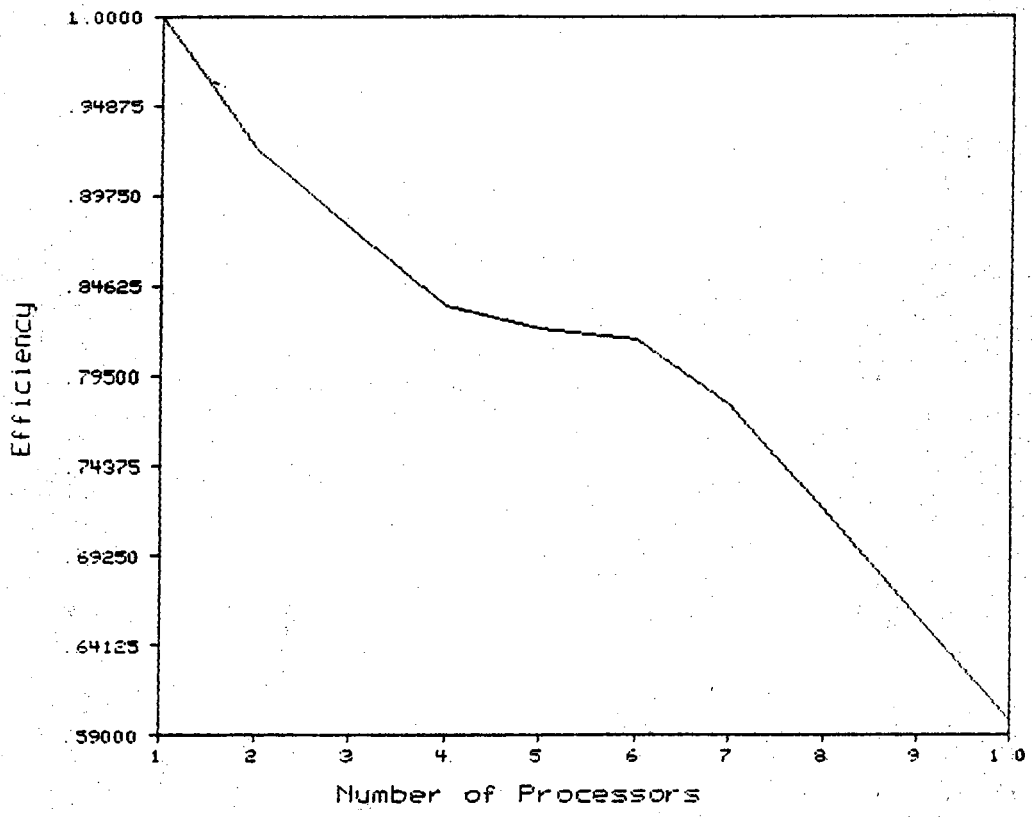
(b): Speedup versus Number of Processors

Fig. 2.3: Simulation Results for Robot Inverse Dynamics Computation on Proposed Architectural Model

The MC 68020 microprocessor is used as the basis of simulation, i.e. each PE_i in Fig. 2.1 is represented by the specified processor type. Table 2.6 specifies particular instruction execution times of a 16.7MHz MC 68020 processor relevant to the processing of the specified problem. This data is used to simulate the performance of the N-E dynamics algorithm on the proposed architecture, and also for the purpose of comparison with previous work.

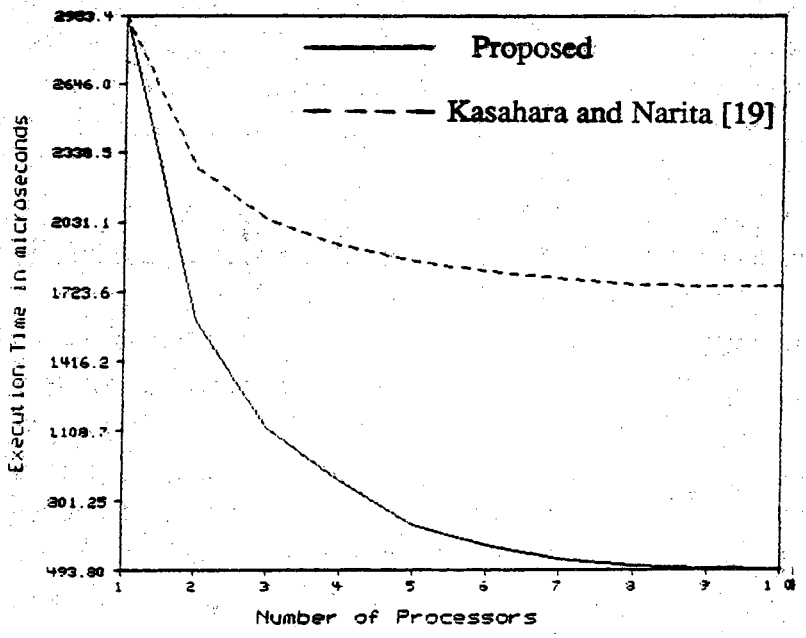
Fig. 2.3(a) shows a plot of the execution time for computing the inverse dynamics problem of a six link robot versus the number of parallel processing elements when using the proposed architecture and scheduling algorithm. An average lower bound processing time of 513.84μs is achieved for 7 to 9 PEs. Note that the time lower bound is not achieved at exactly six parallel processors as required by the ideal case (see Theorem 2.1) when a manipulator with six degrees of freedom is used. In fact, the actual lower bound cost is achieved for the case of nine PEs where the execution time is 100μs lower than that of the ideal case at 6 parallel processors. This slight distortion from the ideal case is due to the irregular number of propagation delays when passing intermediate results through the interprocessor network. As seen in Fig. 2.3(b), speedup>5.7 over the uniprocessor solution is accomplished for $m \geq 8$, where m is the number of parallel processors. Also, Fig. 2.3(c) illustrates the variation of architectural efficiency with the number of parallel processors. The efficiency is defined as S_m/m, where S_m is the speedup for m parallel processors. Intuitively, the efficiency simply gives a measure of how the achieved speedup compares with the ideal speedup. Thus, a high performance evaluation weight should not be given to this plot. From these figures, it is evident that lower bound solutions may be achieved by using m=8, which is also the desired number of PEs if commercially available 8×8 crossbar switches are to be used in the architecture.

We shall now compare the performance of the proposed mutiprocessor architecture and scheduling algorithm for the computation of the inverse dynamics problem to that of Kasahara and Narita [19]. Note that the simulations for comparison purposes will use common MC 68020 processing times of Table 2.6 to maintain fairness. The specified authors use the DFIHS (Depth-First-Implicit- Heuristic-Search) scheduling algorithm (see appendix E) to parallel process the N-E dynamics Equations on a multiprocessor system with a shared single bus interprocessor communication mechanism. Fig. 2.4(a) compares the execution time for the computation of the specified algorithm on the two architectural models when m is varied. A performance improvement of more than 1.75ms for $m \geq 8$ is achieved for the case of the proposed system over that of Kasahara and Narita. In addition, Fig. 2.4(b) verifies that the proposed architecture performs at speedup factors greater than

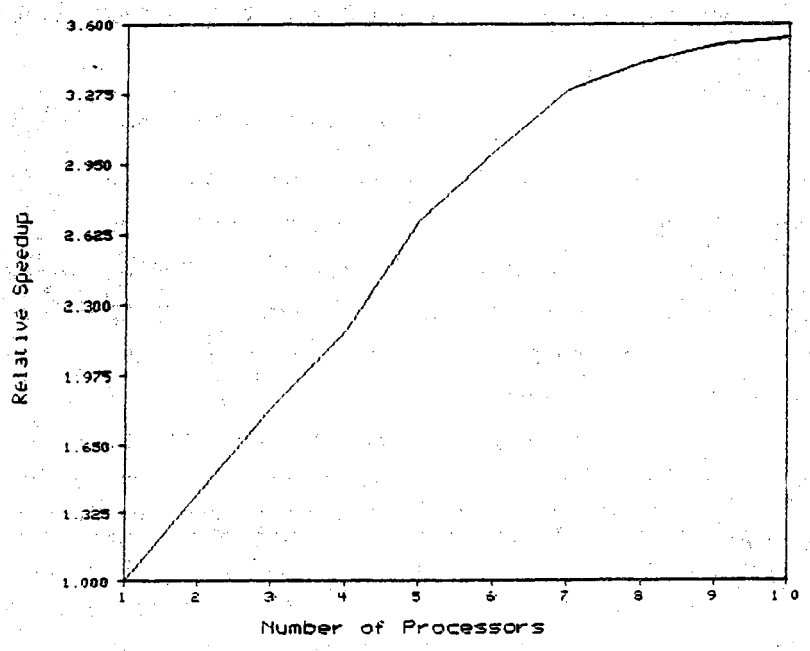


(c): Efficiency versus Number of Processors

Fig. 2.3 (Continued)

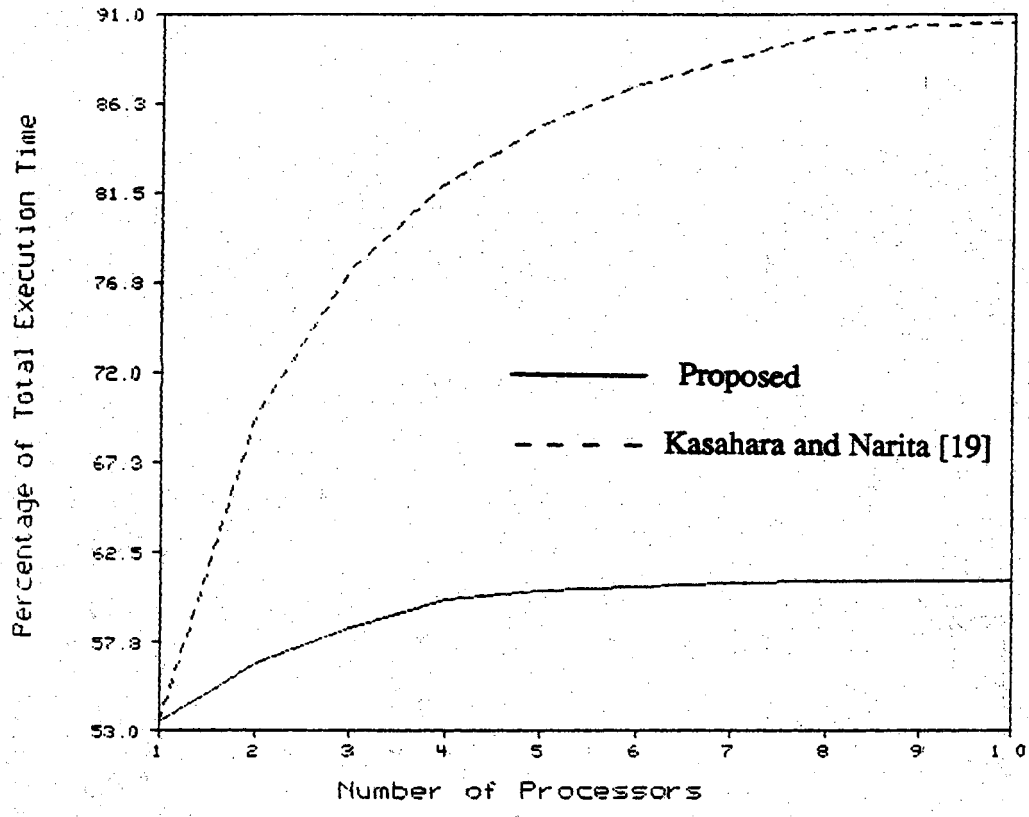


(a): Execution Time versus Number of Processors



(b): Relative Speedup versus Number of Processors

Fig. 2.4: Performance Comparison of Proposed Architecture to that of [19] for Robot Inverse Dynamics Computation



(c): Percentage of Total Execution Time spent on Interprocessor Communication versus Number of Processors

Fig. 2.4 (Continued)

3.4 for m≥8 over the previous architectural model. Fig. 2.4(c) suggests a valid cause for the much lower performance of Kasahara and Narita's parallel processing environment. It is seen that interprocessor communication is a major bottleneck in their shared bus multiprocessor system. In fact, over 90% of the algorithm execution time in their architecture is spent by performing dialogue between PEs for m≥8. Latencies due to arbitration costs for access to the shared bus is thus a crucial performance degradation factor when multiple PEs attempt to perform interprocessor communication via mailboxes at the end of instruction processing cycles. Note the moderately lower percentage of total execution time that the proposed system spends for interprocessor communication (approximately 60% for m≥8). The crossbar interprocessor communication network is naturally one of the most important reasons for the higher perforamnce of the proposed architecture. In addition to optimal scheduling of data through the crossbar network to destination PEs in minimal number of interprocessor transfer latencies, simultaneous operand fetches by the m parallel PEs from respective PEM_i is performed to minimize overall algorithm execution time.

2.3 Parallel Processing of PUMA Forward and Inverse Kinematics

2.3.1 Multiprocessor Architectural Model

The non-linear, non-recursive and irregular form of PUMA forward and inverse kinematics Equations (see Appendix C and D respectively) suggest that these algorithms are not amenable for a SIMD parallel processing environment. In fact, if such a type of architectural model is used for this application, severe performance degradation will occur due to multiple idle/masked processors at each parallel execution stage. The masking of SIMD PEs occur since insufficient number of of identical tasks will be active at any processing level due to the non-recursive nature of the specified algorithms. An alternate architectural approach must therefore be investigated. A multiprocessor system with shared memory for interprocessor communication is studied for this application. This architectural model is illustrated in Fig. 2.5. The memory arbitration mechanism allows access to the mailboxes in shared memory for interprocessor communication on a first-come first-serve basis. The DFIHS (Depth-First-Implicit-Heuristic-Search) scheduling algorithm (see Appendix E) is used to create the static schedule for parallel processing of forward and inverse kinematics equations. This static schedule of parallel tasks is stored in the microcode ROM_i at run time.

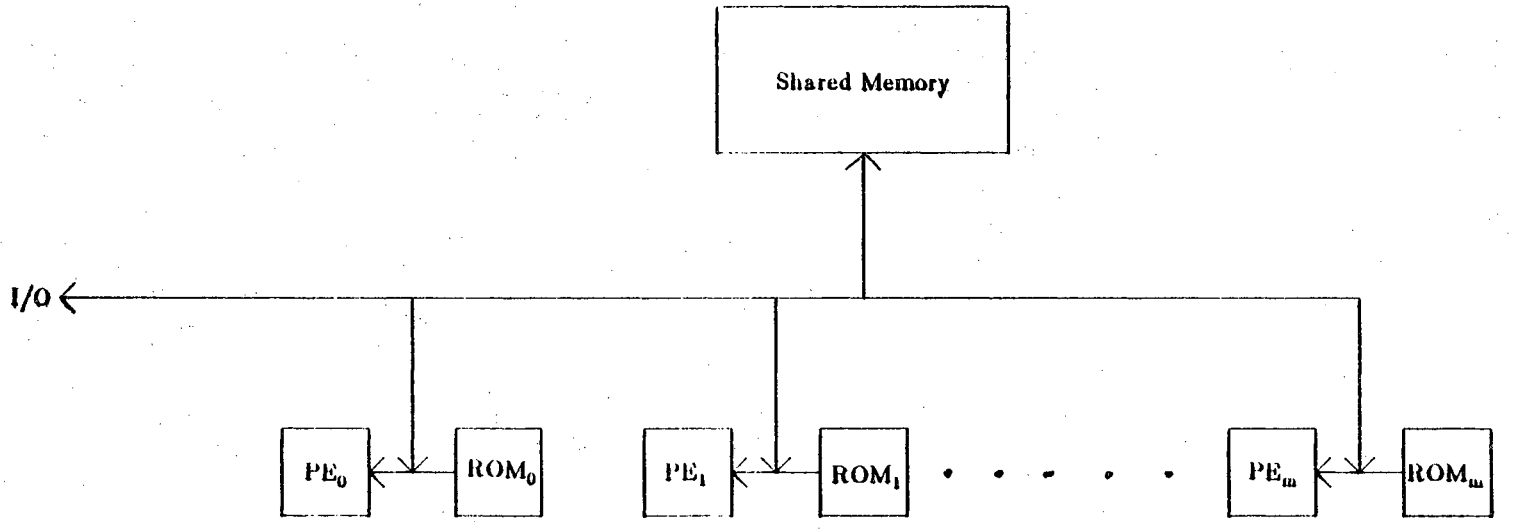
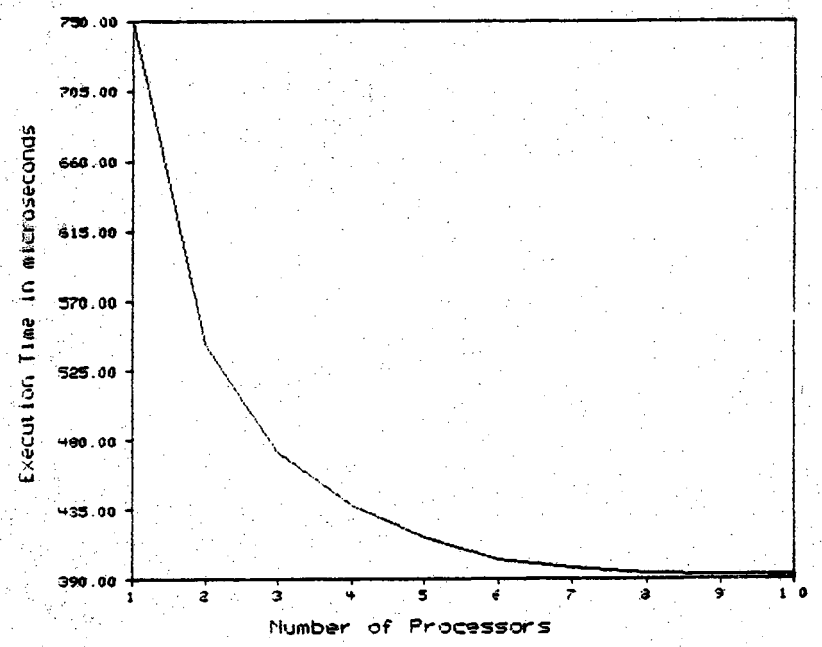


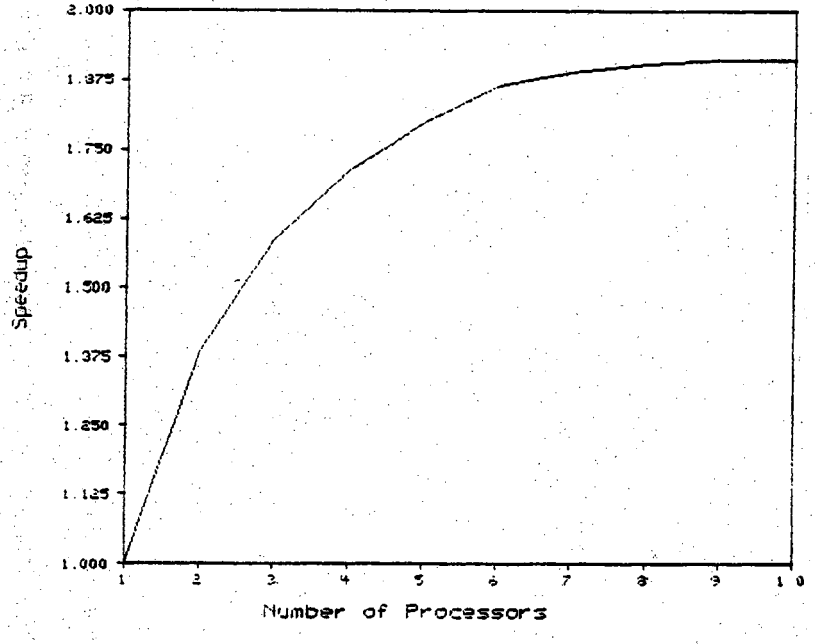
Fig. 2.5: Multiprocessor Architectural Model for PUMA Forward and Inverse Kinematics Computations

Table 2.7: MC 68881 Floating-Point Operation Execution Times

Function	Assembler Code	Operational Delay (in μs)
Fetch Single Operand from Shared M	FMOVE <EA>,FPy	2.28
Fetch Pair of Operands from Shared M	FMOVE <EA>,FPx FMOVE <EA>,FPy	4.55
Store Result in Shared M	FMOVE FPy,<EA>	2.28
Add	FADD FPx,FPy	3.05
Subtract	FSUB FPx,FPy	3.05
Multiply	FMUL FPx,FPy	4.25
Divide	FDIV FPx,FPy	6.17
Square Root	FSQRT FPy	6.41
Arc Tangent	FATAN FPy	24.1
Sine	FSIN FPy	23.4
Cosine	FCOS FPy	23.4



(a) Execution Time versus Number of Processors



(b) Speedup versus Number of Processors

Fig. 2.6: Simulation Results for Forward Kinematics

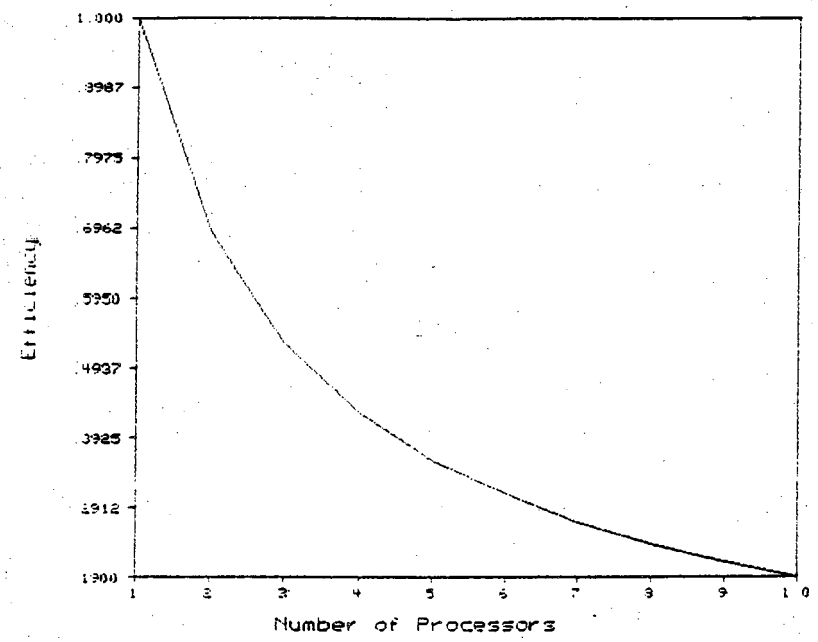
Performance degradation occurs in this architecture due the operational delay caused by multiple PEs waiting for access to shared resources. Note that overall execution speed may not be improved for computation of the specified problems by using an iterprocessor network with local PEM_is since these types of networks are generally only efficient for SIMD machines.

2.3.2 Simulation Results

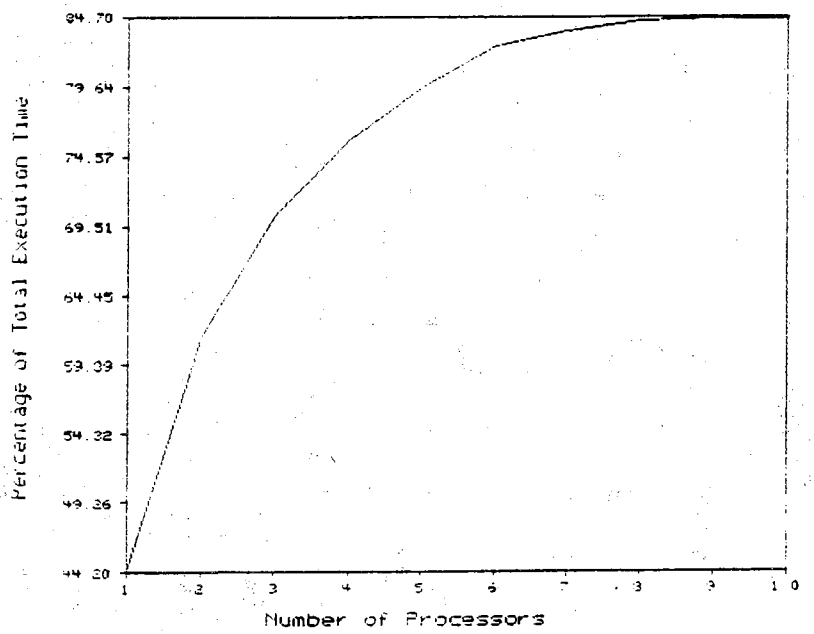
Simulation results for the computation of PUMA forward and inverse kinematics algorithms using the shared memory interprocessor communication strategy and DFIHS task scheduling algorithm (see Appendix E) is presented here. The MC 68020 with its MC 68881 floating point math coprocessor is used as the basis of performance evaluation. In other words, each PE_i in Fig. 2.5 is represented by this dual processor combination. Table 2.7 specifies the execution time data of MC 68881 operations needed for the computation of the specified problems. These processing times are used to generate the simulation results presented hereafter.

The variation of execution time for the forward kinematics algorithm with the number of parallel processors m is depicted in Fig. 2.6(a). Minimum execution time is achieved at approximately 392μs for m≥8. As seen from Fig. 2.6(b), speedup close to 2 is reached for m≥8 over the uniprocessor solution. Also, Fig. 2.6(c) specifies how the architectural efficiency varies with the number of parallel processors. As a reminder, high importance should not be given to this plot since it simply shows how the achieved speedup compares to the ideal speedup. Finally, Fig. 2.6(d) shows the percentage of total execution time that the parallel processors spend for interprocessor communication to evaluate the specified algorithm. Approximately 85% of the overall execution time is spent performing interprocessor communication. This high rate is primarily due to arbitration costs for access to shared memory. Note again that performance will not be improved by using an interprocessor network with SIMD instruction processing, although the interprocessor communication latencies may be lower for this case.

Simulation results for parallel processing of the inverse kinematics algorithm will now be discussed. Fig. 2.7(a) verifies that minimum execution time of approximately 750μs is achieved when five parallel processors are used. Further, speedup of 2.3 occurs for m≥5 over the uniprocessor solution as illustrated in Fig. 2.7(b). For consistency, the variation of efficiency with m is delineated in Fig. 2.7(c). Finally, Fig. 2.7(d) confirms that over 61% of total algorithm execution time is spent for interprocessor communication through shared memory. Notice that this last result is considerably lower than the percentage of time the forward kinematics algorithm

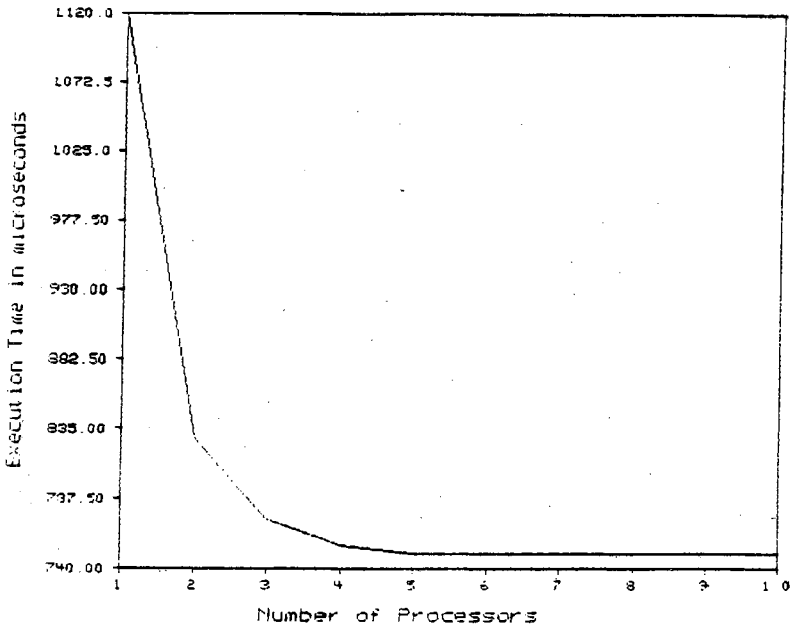


(c) Efficiency versus Number of Processors

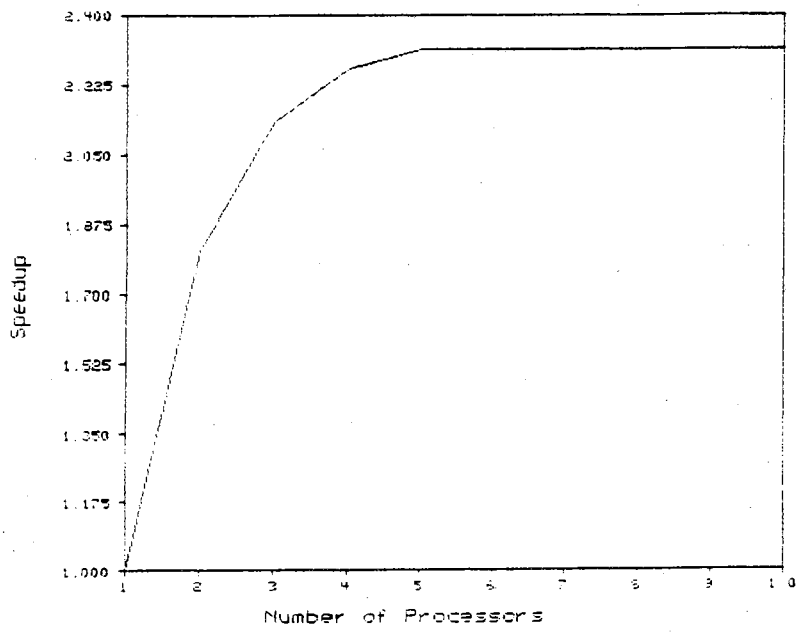


(d): Percentage of Total Execution Time spent on Interprocessor Communication versus Number of Processors

Fig. 2.6 (Continued)

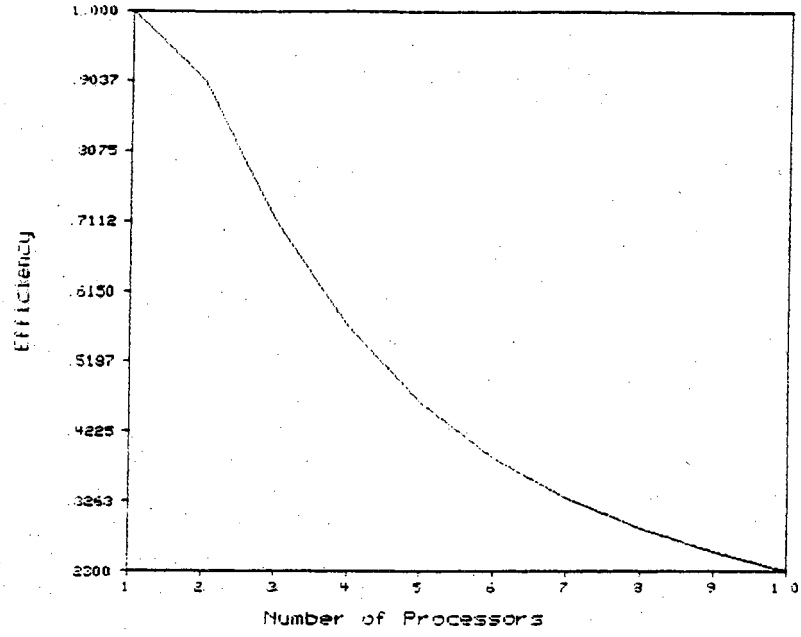


(a): Execution Time versus Number of Processors

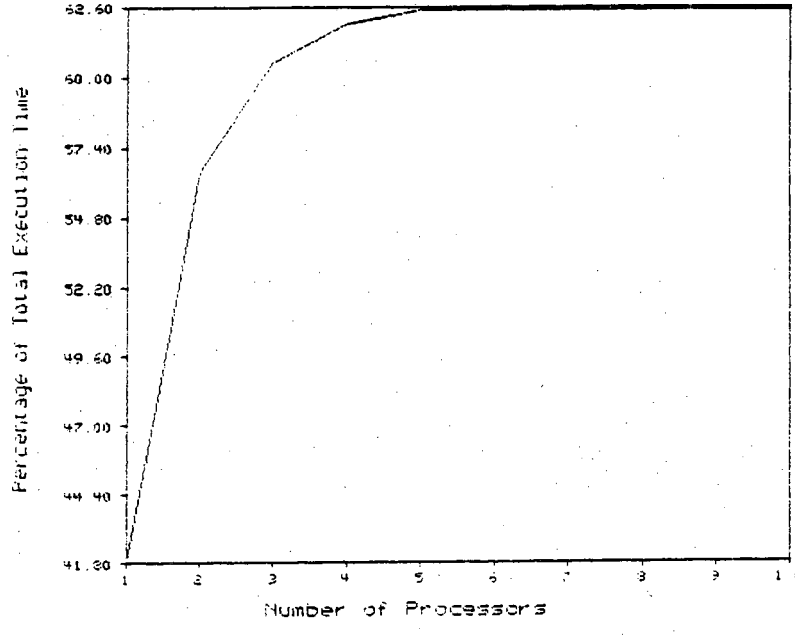


(b): Speedup versus Number of Processors

Fig. 2.7: Simulation Results for Inverse Kinematics



(c): Efficiency versus Number of Processors



(d): Percentage of Total Execution Time spent on Interprocessor Communication versus Number of Processors

Fig. 2.7 (Continued)

spends for interprocessor communication.

2.4 Conclusions

The ideal lower bounds on the number of parallel processors and execution time to evaluate the inverse dynamics problem of an n-link manipulator in a parallel processing architecture using an optimal scheduling algorithm was derived to be n and $o(\lceil a_1 n \rceil)$ respectively, where a_1 is a specified constant. Next, a novel SIMD scheduling algorithm to perform parallel processing on a SIMD multiprocessor-based architectural model with a crossbar interprocessor network to compute the specified problem close to the ideal lower bound is presented. The performance of the N-E dynamics algorithm on the proposed architecture using the specified scheduling technique is next simulated with the MC 68020 representing each processing element. Speedup factor of greater than 3.4 over previous related work [19] for the computation of the defined problem on a parallel processing system is achieved. The SIMD architectural model is then not used again to simulate the performance of PUMA forward and inverse kinematics algorithms because of the non-recursive and non-linear nature of these algorithms. A multiprocessor model with a shared memory interprocessor communication strategy is instead investigated for this purpose using the DFIHS scheduling algorithm. The MC 68020 with its MC 68881 math coprocessor is used as the basis of the next few simulations to determine the performance of the kinematics algorithms on the specified architecture. Results show speedup of approximately 2 over the uniprocessor solution when the number of parallel processors is greater than eight and five for the cases of forward and inverse kinematics algorithms respectively. Very large speedups is not achieved in this latter multiprocessor architectural model due to high interprocessor communication latency costs.

CHAPTER 3
A COST EFFICIENT BIT-SERIAL ARCHITECTURE FOR ROBOT INVERSE DYNAMICS COMPUTATION

3.1 Introduction

The parallel, pipelined, and recursive nature of the N-E dynamics algorithm suggests that it is amenable to a custom bit-serial array architecture. This chapter presents such an architecture to compute the desired algorithm within the bit-serial execution time lower bound of $o(\lceil c_1k + c_2kn \rceil)$, where c_1 and c_2 are specified constants, k is the system word length, and n is the number of manipulator links. A multi-functional bit-serial cell is developed as a building block for the recursive array unit proposed. The cell's fault-tolerant external asynchronous communication protocol and its high-performance design using Zipper CMOS [10] circuit structures is of particular interest. This cell may easily be used as a "standard cell" to design the two array chips which realize the forward and backward recursions, respectively, of the N-E dynamics formulation. In addition to these two chips, a minimum number of external FIFO register files are required to implement the proposed system.

3.2 Choice of a Bit-Serial Architecture

The Newton-Euler dynamics formulation is a highly pipelined recursive process. A bit-serial architecture is therefore a natural candidate for its efficient computation in a closely pipelined cost-efficient manner. The single wire input and output of bit-serial devices allows tight pipelining of operational cells, leading to efficient communication within and between chips. This is an important advantage since communication issues are a dominant factor in the computation of the inverse dynamics problem. In addition, when comparing *n*-bit word length bit-serial and bit-parallel arithmetic operators, we can expect the bit-serial part to contain 1/*n*th of the hardware of the bit-parallel device [8]. Minimum chip count solutions are therefore

feasible in bit-serial systems without the problems of on-chip routing of data lines. In fact, the N-E dynamics algorithm may easily be computed in the proposed architecture with only two bit-serial chips in addition to a minimum number of external FIFO register files. Preliminary studies show the two chips require a combined die area of approximately $70mm^2$ when fabricated in a 1.2μ CMOS technology, using the proposed bit-serial cell architecture as the standard building block. If our proposed architecture is implemented using a bit-parallel architecture, a total of 32 matrix processors capable of performing matrix-vector multiplications and vector cross-products operations would be required. In addition, a more complex control network is needed for the bit-parallel case. Furthermore, the parallel wiring interconnections between bit-parallel processors suggests that it is a cost-inefficient approach.

3.3 Time Lower Bound to Compute the Inverse Dynamics Problem on a Bit-Serial Architecture

In this section we will discuss the limitation of speeding up the inverse dynamics computational problem when running on a bit-serial system. Before deriving the time lower bound, the following notation and lemma must be established.

Notation:

(1) E<l> denotes a linear arithmetic expression E of l distinct atoms, where an atom is a constant or a variable, e.g. E<4>=a+b–c/d.

(2) T_k = Minimum time to evaluate an expression *E<l>* in a non-recursive bit-serial system.

(3) T_1 = Shortest time to compute the first iteration of the N-E algorithm in a bit-serial system.

(4) T_{n-1} = Shortest time to evaluate each additional iteration when computing the remaining n–1 iterations of the N-E algorithm in a recursive bit-serial system.

(5) T_n = Minimum time to compute the inverse dynamics problem in a recursive bit-serial system.

Lemma 1: The time lower bound of $T_k[E<l>]$ [8]. The shortest bit-serial processing time to evaluate an arithmetic expression E<l> is bounded below by and equal to $o(\lfloor k \rfloor)$, in other words,

$$T_k[E<l>] \geq o(\lfloor k \rfloor)$$

Theorem 1: The minimum time to compute the inverse dynamics problem of an *n*-link manipulator in a recursive bit-serial architecture is bounded below by $o(c_1\lfloor k \rfloor + c_2\lfloor kn \rfloor)$, where k is the system word length, c_1 and c_2 are specified constants.

Proof: Addition, subtraction and multiplication are the primitive arithmetic operations prevalent in any iteration of the inverse dynamics problem. By beginning a subsequent operation before finishing the previous one, a bit-serial arithmetic cell requires k clock cycles to evaluate any one of the three specified operations. The minimum time to compute the first iteration of the N-E algorithm may therefore be expressed as a function containing at least *3k* atoms. Using Lemma 1, we thus have,

$$T_1[E<3k>] \geq O(\lfloor 3k \rfloor). \tag{1}$$

Let b_1k (where b_1 is a constant) be an uniform clock period count needed to evaluate each additional iteration of the N-E algorithm in a recursive bit-serial system. To better conceptualize, b_1k clock periods is the constant additional time required to compute those operations in the ith iteration which are not evaluated in parallel to the (i−1)th computational cycle. Therefore, the additional time to calculate the remaining (n−1) iterations may be expressed as follows:

$$T_{n-1} = b_1k(n-1).$$

Clearly, the minimum time to determine T_{n-1} is when b_1 is one. Thus, the lower bound of T_{n-1} is defined as,

$$T_{n-1}[<n-1>] \geq o(\lfloor k(n-1) \rfloor). \tag{2}$$

By combining equations (1) and (2), we obtain

$$T_n[E<n>] \geq o(\lfloor 2k + kn \rfloor).$$

The inverse dynamics problem may be considered as computing a set of arithmetic operations which result in obtaining the joint torques. Each joint torque τ_i of joint i can be expressed as an arithmetic operation containing at

least 3n atoms: n joint positions $\{q_i\}_{i=1}^{n}$, n joint velocities $\{\dot{q}_i\}_{i=1}^{n}$, and n joint accelerations $\{\ddot{q}_i\}_{i=1}^{n}$, implying,

$$T_n[E<3n>] \geq o(\lfloor 2k + 3kn \rfloor). \tag{3}$$

Rewriting equation (3) without changing the order of the lower bound to evaluate a set of *n* torques,

$$T_n[\tau_1,\tau_2,...,\tau_n] \geq o(\left\lfloor c_1k + c_2kn \right\rfloor).$$

where c_1 and c_2 are specified constants. Q.E.D.

By establishing a time lower bound in Theorem 1, we can now compare and contrast bit-serial computational structures by simply comparing the coefficients c_1 and c_2. It should be noted here that such comparisons will only be valid if the competing systems use the same word length and manipulator link count.

3.4 Bit-Serial System Architecture

The high-level view of the bit-serial system architecture is illustrated in Fig. 3.1. Basically, the core of the system consists of two bit-serial array processors: Processor 1 computes the forward recursions and Processor 2 evaluates the backward iterations of the N-E algorithm.

Operand feeding of the bit-serial arrays is an asynchronous mechanism. Initially, the host processor loads the rotation (${}_{i}^{i+1}R$) and inertia (${}^{i}I_i$) matrices, relative position (${}^{i}P_{i+1}$), joint velocity ($\dot{\theta}$) and acceleration ($\ddot{\theta}$) vectors, and the scalar link masses (m_i) into the PISO FIFO (Parallel-In-Serial-Out First-In-First-Out) register blocks in the order shown. The asynchronous handshaking protocol used by the bit-serial cells to communicate with external devices allows computations to be initiated whenever both operands of any bit-serial cell are available. The philosophy employed is similar to that of data flow computer architectures. This eliminates the "initial delay time" (time required to set up *all* the operands) which degrades the performance of many custom algorithmic processors.

Whenever the host processor loads any operand(s) into the FIFO, it also informs the respective bit-serial cell whose operands are currently the load has been performed. The notified cell subsequently provides the clock pulses to its operand FIFOs to load its input data synchronously in a bit-serial fashion. Pulses delivered by individual cells for operand loading enhances fault tolerance by eliminating any

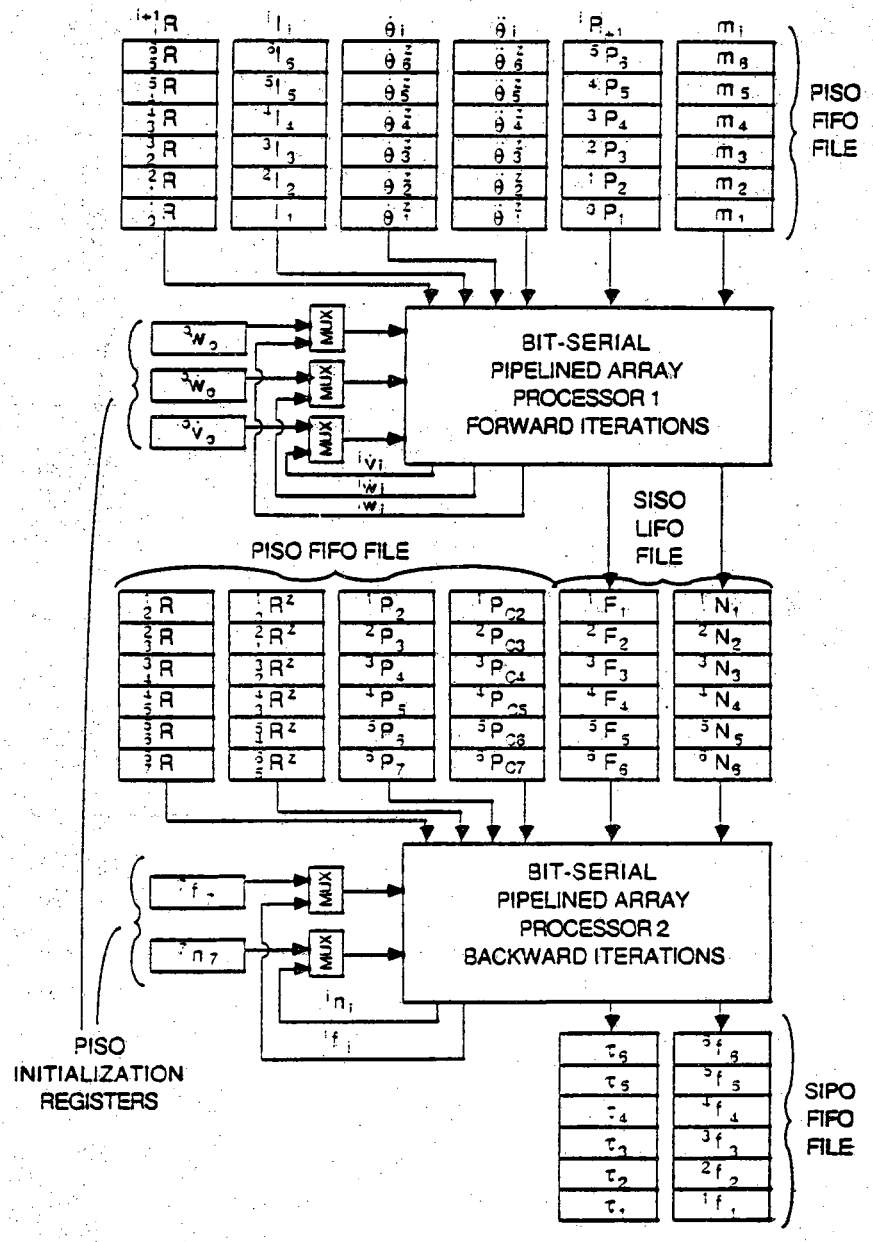


Fig. 3.1: Bit-Serial System Architecture

possibility of misalignment of multi-operand data bits.

Array processor 1 computes the angular velocity ($^i\omega_i$) and acceleration ($^i\dot{\omega}_i$), and the linear acceleration ($^i\dot{v}_i$) vectors which are fed back multiplexed with their initialization vectors as inputs to cells earlier in the processor's pipeline. Outputs of the processor are total external forces (^iF_i) and moments (^iN_i) exerted at the center of mass of link i. These computed results are then loaded into array processor 2 through the SISO LIFO (Serial-In-Serial-Out Last-In-First-Out) register files. A LIFO structure is used since the total forces and moments computed in processor 1's last iteration are needed as operands of processor 2's first iteration.

Processor 2 initiates execution at the end of processor 1's nth iteration, where n is the number of manipulator links. Operands are loaded into this processor in a manner identical to Processor 1. Intermediate results of computations in the processor include forces (^if_i) and moments (^in_i) exerted on link i by link (i–1) with respect to the base frame. These results are subsequently fed back multiplexed into the array with their initialization vectors. A complete set of joint torques and forces (τ_i) are obtained at the end of the nth backward iteration and loaded into an SIPO FIFO (Serial-In-Parallel-Out First-In-First-Out) register file as system outputs.

3.5 Bit-Serial Array Processor Organization and Performance

At this stage in the paper, we shall assume that an individual bit-serial cell is capable of computing the 3-D vector and matrix arithmetic operations depicted in Appendix B. Details on how the cell implements these functions is discussed in Section 3.7.

Fig. 3.2 illustrates the parallel and pipelined organization of bit-serial cells in Processor 1 to evaluate the forward iterations of the N-E algorithm. The initials within each node denote the type of operation that the particular cell is required to perform. Also, the number next to each cell serves as a means of individual cell identification. Interconnections between cells represent "single-wire" bit-serial data paths. The cells are organized in eight stages of a pipeline with some intermediate results being fed back as inputs to cells earlier in the pipeline in a recursive fashion.

Table 3.1 summarizes the computation times of the various matrix and vector arithmetic functions which may be evaluated by any individual bit-serial cell. The operational delay data from this table is used to construct the pipeline schedule of tasks to compute the first three iterations of the N-E dynamics algorithm in Processor 1 (Table 3.2). The unit on the time scale of Table 3.2 is k clock periods, where k is

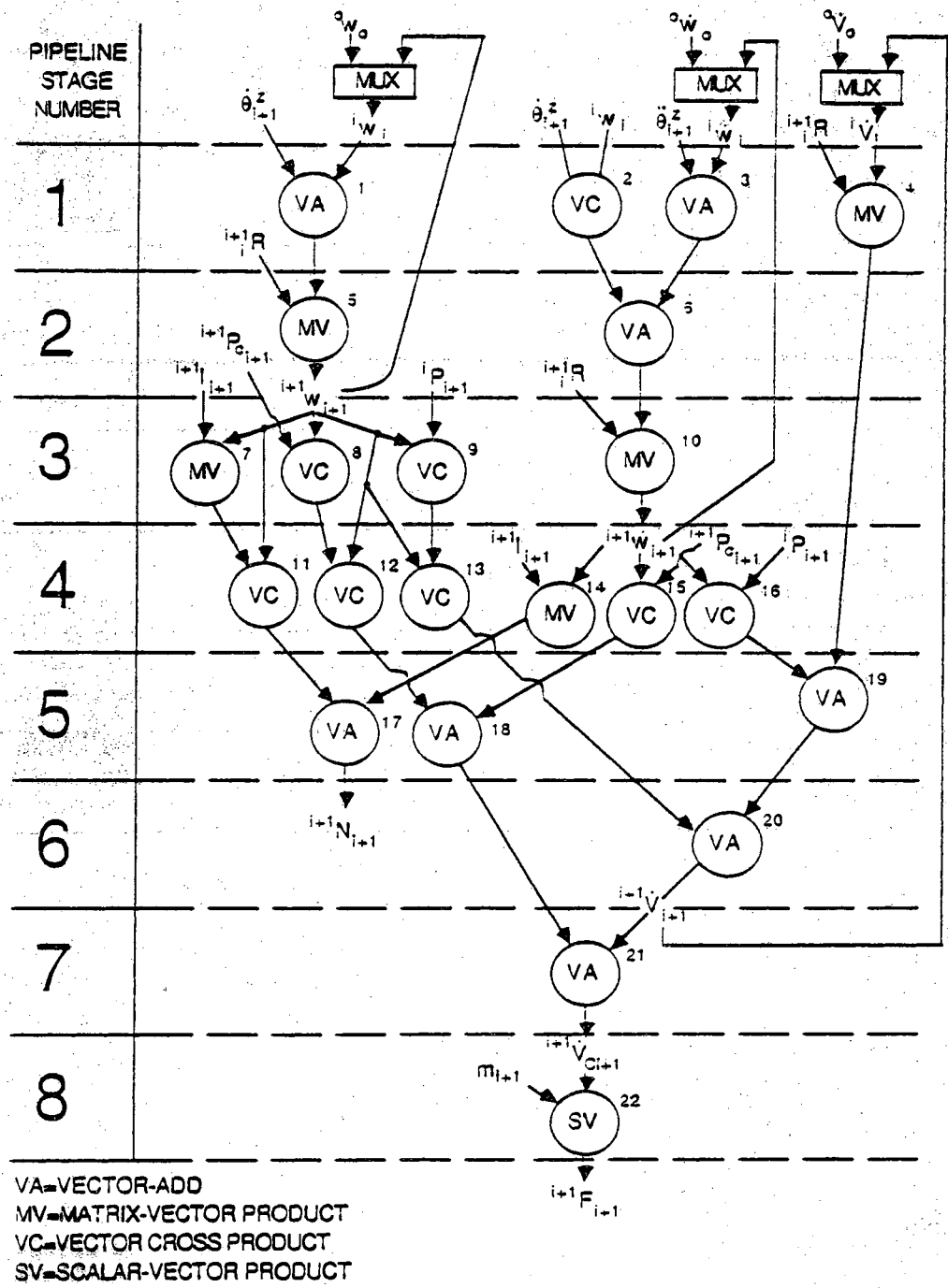


Fig. 3.2: Pipelined Organization of Bit-Serial Cells in Processor 1

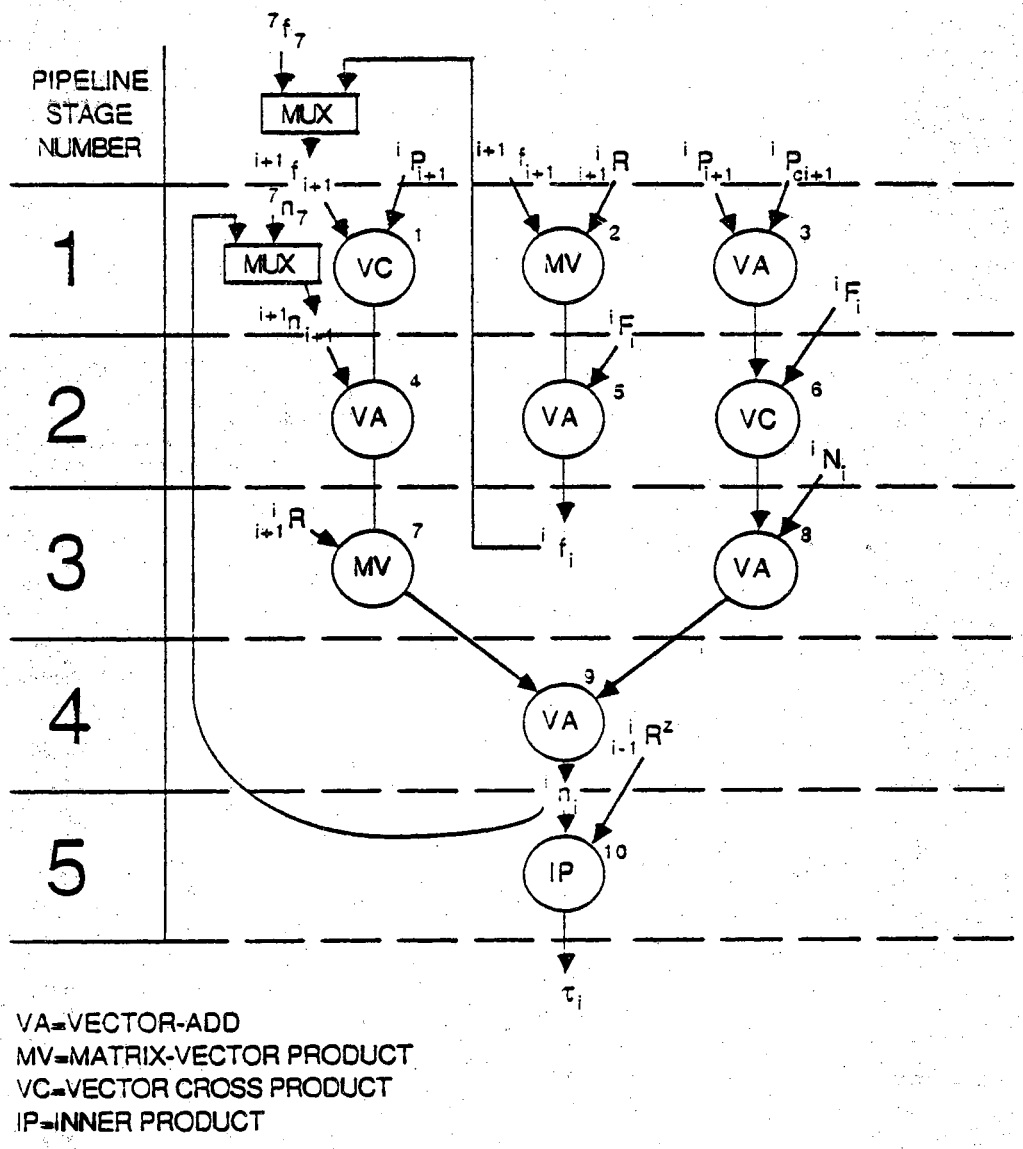


Fig. 3.3: Pipelined Organization of Bit-Serial Cells in Processor 2

Table 3.1: Bit-Serial Cell Performance Measures

Operation	No. of Cells Required	Individual Cell Utilization	Operation Delay (In # of Clock Cycles)	Operation Delay Using 16-Bit Wordlength and Clock Frequency of 20-MHz
Vector-Add	1	3 Adders	k	800 ns
Scalar-Vector Product	1	3 Multipliers	k	800 ns
Inner Product	1	3 Multipliers 2 Adders	2k	1.6μs
Matrix-Vector Product	3	3 Multipliers 2 Adders	2k	1.6 μs
Vector Cross-Product	3	3 Multipliers/ 3 Subtractors	2k	1.6 μs

the system word length. Also, the numbers in the iteration columns indicate which bit-serial cell of Fig. 3.2 is executing a particular operation during the specified time interval. Notice that bit-serial cells begin execution of tasks in successive iterations as soon as they complete their designated operations in the previous iteration, provided process precedence relations still hold true. Thus, the parallelism between successive iterations allows high cell utilization. This higher performance operational feature is supported in a fault-tolerant manner by the asynchronous handshaking protocol (described in Section 3.7) used by the bit-serial cells for external communication. A total of 11k clock periods are needed to compute the first forward iteration. The massive amount of parallelism between tasks of successive iterations allow each additional iteration to be processed in only 4k clock cycles.

Similarly, Fig. 3.3 illustrates the organization of bit-serial cells in array processor 2 to compute the backward iterations. Accordingly, Table 3.3 depicts its pipeline schedule to process three backward iterations. The first backward iteration requires 8k clock periods, whereas each additional iteration needs 2k clock cycles for completion.

The total time to compute the inverse dynamics problem on the proposed architecture may, therefore, be expressed as [19k + 6kn] clock periods, where k is the system word length and n is the number of manipulator links. Thus, for comparison purposes, the coefficients c_1 and c_2 (defined earlier in Section 3.4) are 19 and 6, respectively, to compute the inverse dynamics problem in our particular recursive bit-serial system architecture. The total time to compute the N-E dynamics algorithm is, therefore, 44μs when a 16-bit system word length, 20MHz operating frequency, and a 6-link manipulator is used.

3.6 Multi-Functional Bit-Serial Leaf Cell Architecture and Operation

In this section, we shall describe the architecture and operation of a multi-functional bit-serial cell capable of performing the various 3-D matrix/vector arithmetic operations depicted in Appendix B. A block diagram of this cell is shown in Fig. 3.4. The 3-bit mode register determines the type of vector operation that the particular cell is required to perform. This register is set by direct hardware control. A PLA-based control unit is used for the control of the external asynchronous communication mechanism and internal operation mode. The counter provides load pulses to operand cells to obtain the cell's input data in a synchronous fashion. Shift registers, "previous" and "new", store up to two intermediate results temporarily when the cell to which these results are to be sent is not ready to receive new operands. The three voters determine bit-serial data routing paths to support alternate arithmetic

Table 3.2: Parallelism in the Execution of Tasks for Three Forward Iterations

Time (In Number of Clock Periods)	Iteration #1	Iteration #2	Iteration #3
k	1 2 3 4		
2k	2 4 5		
3k	5 6	2	
4k	7 8 9 10	1 2	
5k	7 8 9 10	5	2
6k	11 12 13 14 15 16	3 5	2
7k	11 12 13 14 15 16	6 7 8 9	1
8k	17 18 19	7 8 9 10	5
9k	20	10 11 12 13	5
10k	21	4 11 12 13 14 15 16	3 7 8 9
11k	22	4 14 15 16	6 7 8 9
12k		17 18 19	10 11 12 13
13k		20	10 11 12 13
14k		21	4 14 15 16
15k		22	4 14 15 16
16k			17 18 19
17k			20
18k			21
19k			22

Table 3.3: Parallelism in the Execution of Tasks for Three Backward Iterations

Time (In Number of Clock Periods)	Iteration #1	Iteration #2	Iteration #3
k	1 2 3		
2k	1 2 6	3	
3k	4 5 6	2	3
4k	7 8	1 2 6	
5k	7	1 5 6	2
6k	9	4 8	1 2 6
7k	10	7	1 5 6
8k	10	9	4 8
9k		10	7
10k		10	9
11k			10
12k			10

operating modes. The function of the voters is illustrated in Fig. 3.7(a)-(d). It should be noted here that a matrix-vector multiplication task (not shown in Fig. 3.7) is realized by simply combining three inner product cells. Individual cell utilization and operation delay within the cells to evaluate the matrix/vector operations is summarized in Table 3.1.

Fig. 3.5 shows the control and data line interconnections between neighboring bit-serial cells. Asynchronous handshake control signals, RTRIN (Ready-To-Receive-IN), RTROUT (Ready-To-Receive-OUT), RTSIN (Ready-To-Send-IN), RTSOUT (Ready-To-Send-OUT), LOADCLKIN (Load-Clock-IN) and LOADCLKOUT (Load-Clock-OUT) use an inter-cell communication protocol to achieve higher performance and fault-tolerance in the system. The flow chart illustrated in Fig. 3.6 provides details of this asynchronous communication protocol.

When a cell is ready to receive new operands, it indicates this by asserting the RTROUT signal, and if the preceding cells are ready to send the operands, they activate the RTSIN lines. In response, the current cell generates LOADCLKOUT pulses to load its operands in a bit-serial fashion. This particular feature allows synchronized loading of multi-operand data bits. Near the end of an operation, the succeeding cell may have the RTRIN signal asserted, which indicates it is ready to receive its operands. In that case, the current cell activates its RTSOUT line, thus indicating that it is ready to send the operands. Subsequently, the next cell provides the LOADCLKIN pulses to load its input data bit-serially. However, if the succeeding cell has the RTRIN signal negated at the end of any operation, the result will be loaded into temporary "previous/new" shift registers. If both the "previous" and "new" shift registers are full at any point in time, the given cell is not allowed to load any further operands for execution until one of these registers transfers its contents into the succeeding cell.

3.7 Implementation of Bit-Serial Cell In CMOS Technology

Our design strategy utilizes Zipper CMOS [28] circuit structures. It is basically an improved version of Domino [20] and NORA [14] dynamic CMOS circuits, which are characteristically immune to the problems of instability and charge-sharing prevalent in the latter two techniques. Also, area utilization is considerably better and Zipper CMOS circuits can operate at least two times faster than static CMOS circuits. In addition, its high density allows it to be clocked at very high frequencies. It is thus very advantageous to apply this type of circuit design methodology to bit-serial signal processing.

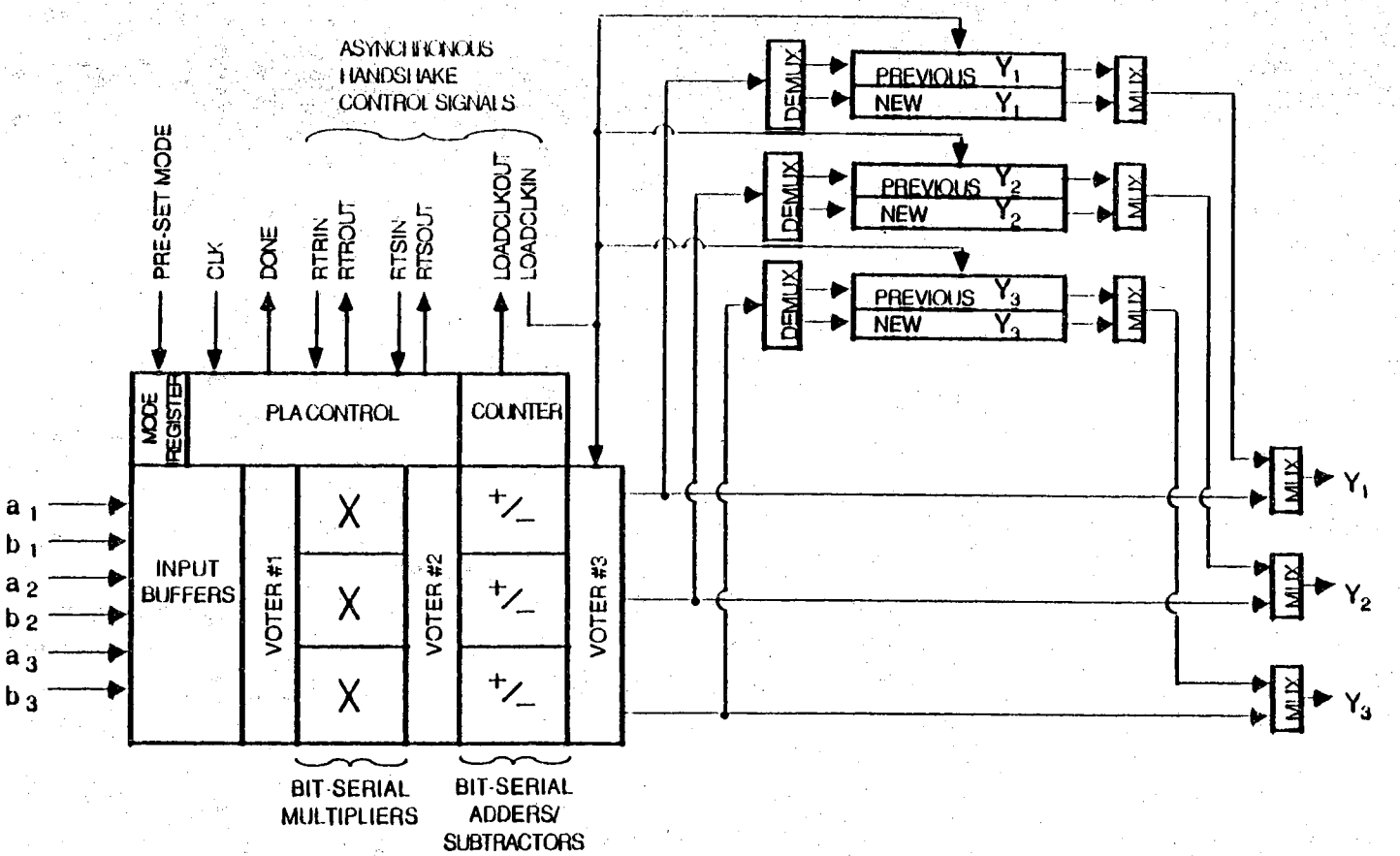


Fig. 3.4: Architecture of a Multi-Functional Bit-Serial Leaf Cell

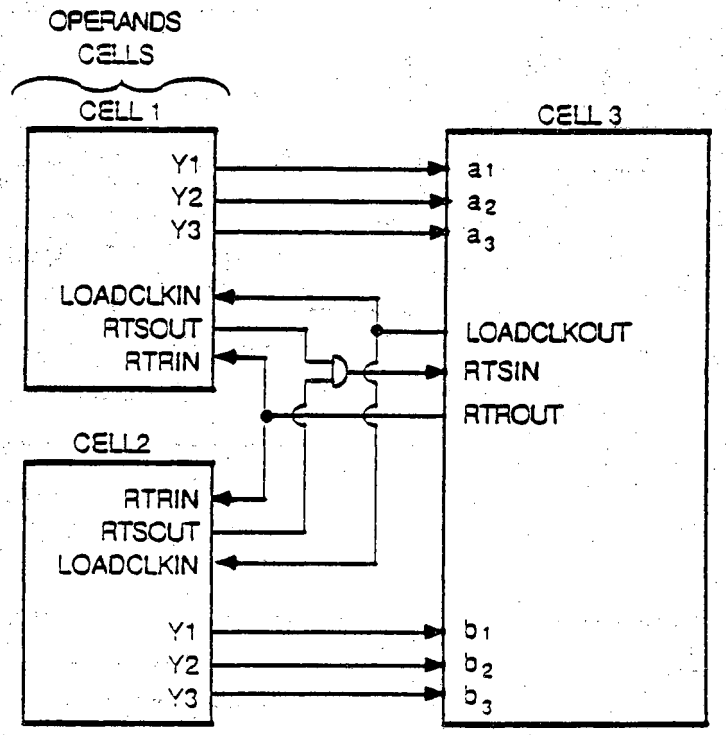


Fig. 3.5: Interconnections Between Neighbouring Bit-Serial Cells

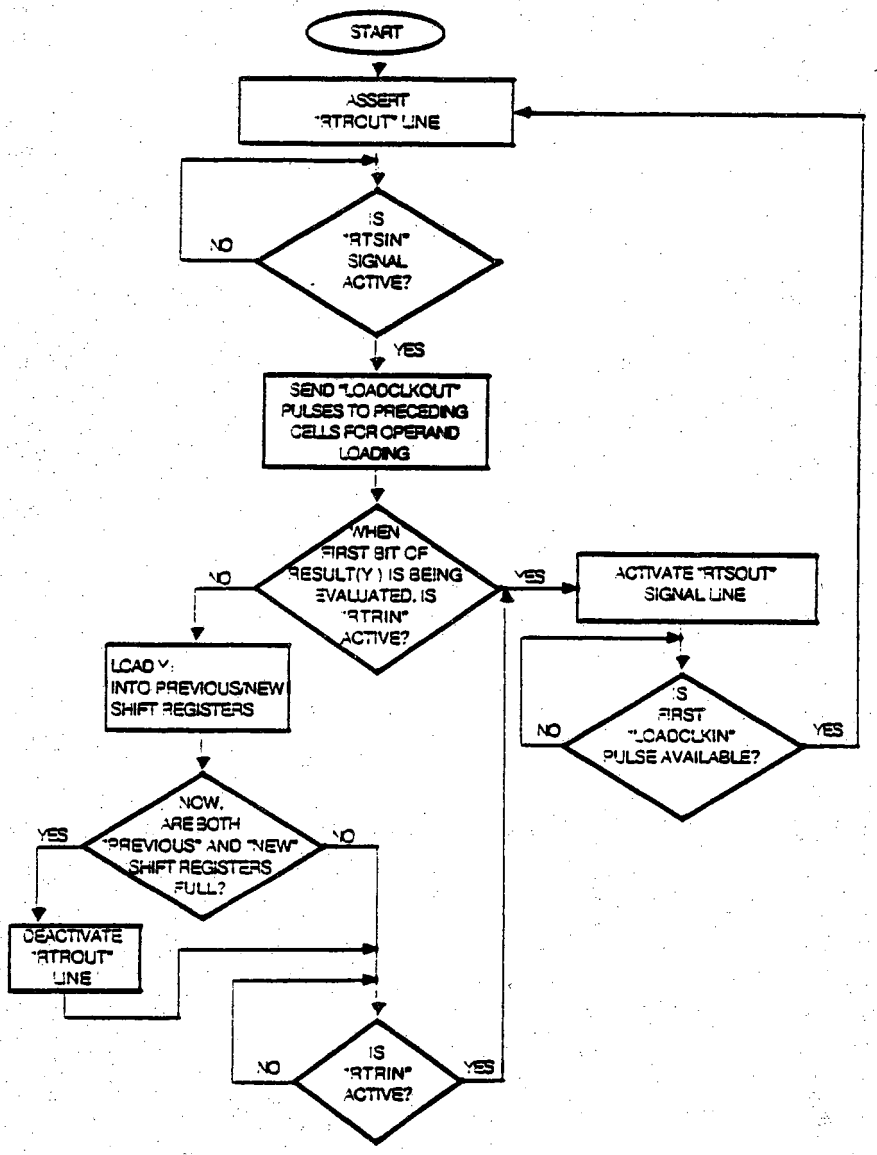
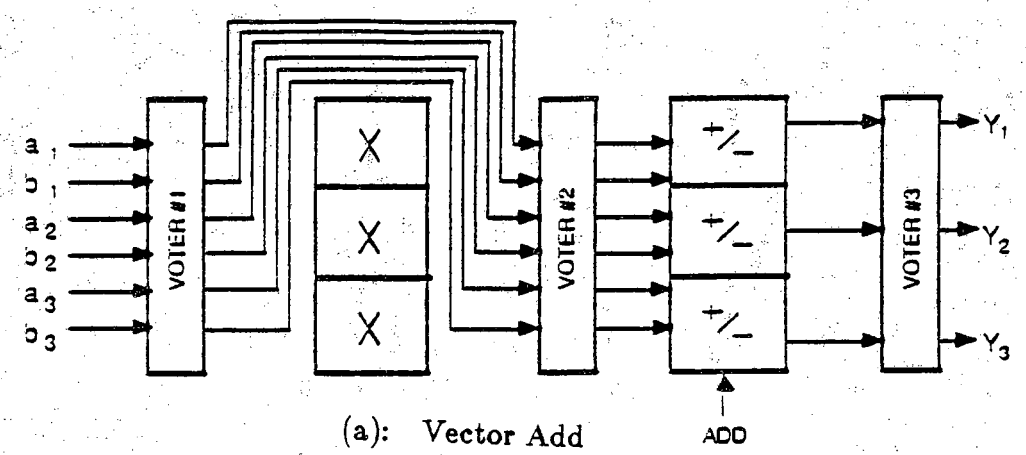
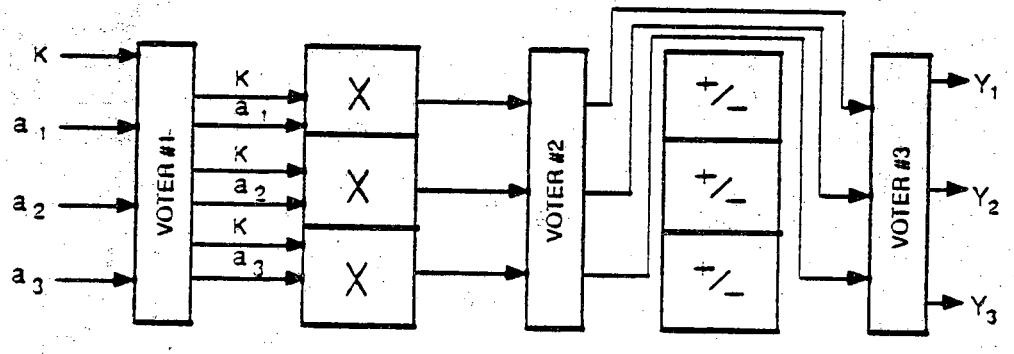


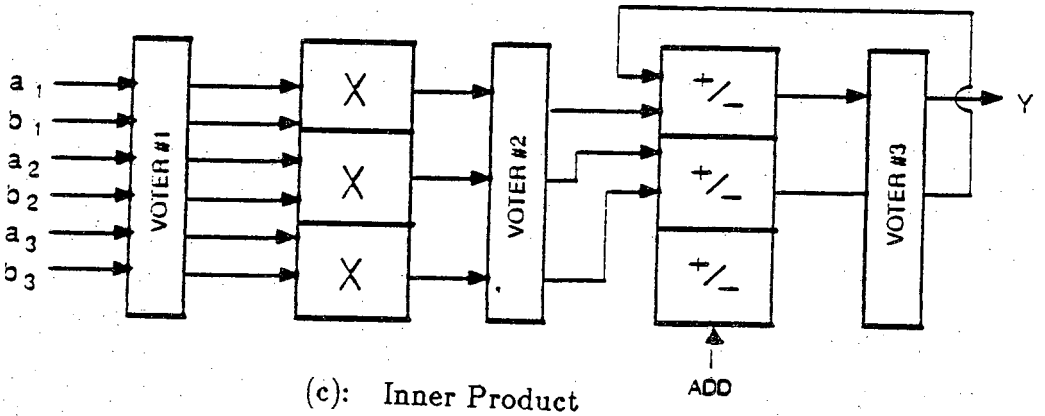
Fig. 3.6: Flowchart of Asynchronous Communication Protocol



(a): Vector Add

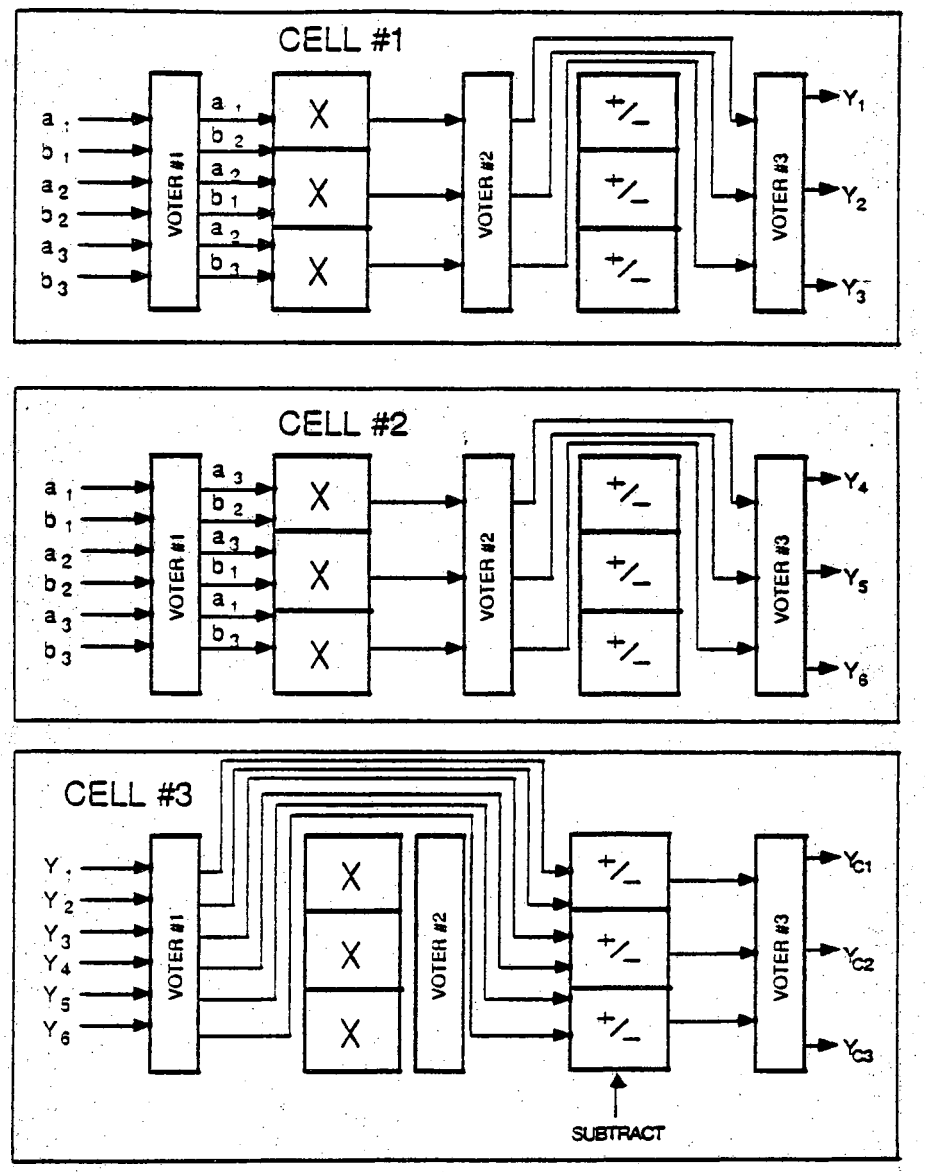


(b): Scalar-Vector Product



(c): Inner Product

Fig. 3.7: Data Routing Paths for Alternate Vector Arithmetic Operating Modes



(d): Vector Cross Multiply

Fig. 3.7 (Continued)

The basic structure of a Zipper CMOS circuit uses a single Zipper driver circuit which generates four strobe signals to drive subsequent N-P CMOS logic blocks. During precharge, the output of every N block is high and that of every P block is low. This ensures that the transistors of each dynamic block are off. During evaluation, the output of each N stage can undergo only one transition, from high to low, and the output of each P stage can undergo only one transition, from low to high. Signals therefore propagate down each stage of the circuit in a "staggered" fashion.

In the Zipper driver circuit of Fig. 3.8, strobe signals ST and $\overline{ST}$ simply act as a two phase clock to drive the logic stages. Signals ST^1 and $\overline{ST}^1$ which feed the N block's precharge pullup and P block's precharge pulldown, respectively, produce a residual voltage which keep the precharge transistors slightly on to overcome leakage currents. It is therefore clear that this driver plays a major role in insuring stability and minimizing charge-sharing effects. It should be noted that only one driver circuit is needed for each bit-serial multiplier and adder/subtractor unit used in the cell.

Fig. 3.9 illustrates our implementation of the bit-serial full-adder/subtractor using Zipper CMOS circuits and dynamic latches. It is designed as a reduced majority function of its inputs. Simulation results have shown the propagation delay of this module to be 37ns. The bit serial adder is also the most significant circuit in the bit-serial multiplier. This suggests that an individual cell may operate at frequencies of up to 25MHz.

The design of a single module of Lyon's 2's complement bit-serial pipelined multiplier in Zipper CMOS is shown in Fig. 3.10. The module is implemented using a Zipper CMOS full-adder, pass-transistors, and dynamic registers. For a k-bit system word length, a bit-serial multiplier consists of k such modules, with serial interconnections of the data words (A_i and B_i), the partial product sum (PPS_i), and the control signal R_i. This latter control signal is used to truncate the low order bit of the partial product sum and to clear the carry flip-flop after each multiplication. The last module of such a multiplier employs a full subtractor instead of a full-adder, as is required by the two's complement algorithm. The bit-serial multiplier takes k clock periods to form a k-bit product of two k-bit operands, by beginning a subsequent operation before finishing the previous one.

3.8 Conclusions

We have described a cost-efficient parallel and pipelined bit-serial system for the inverse dynamics computational problem to achieve the bit-serial execution time

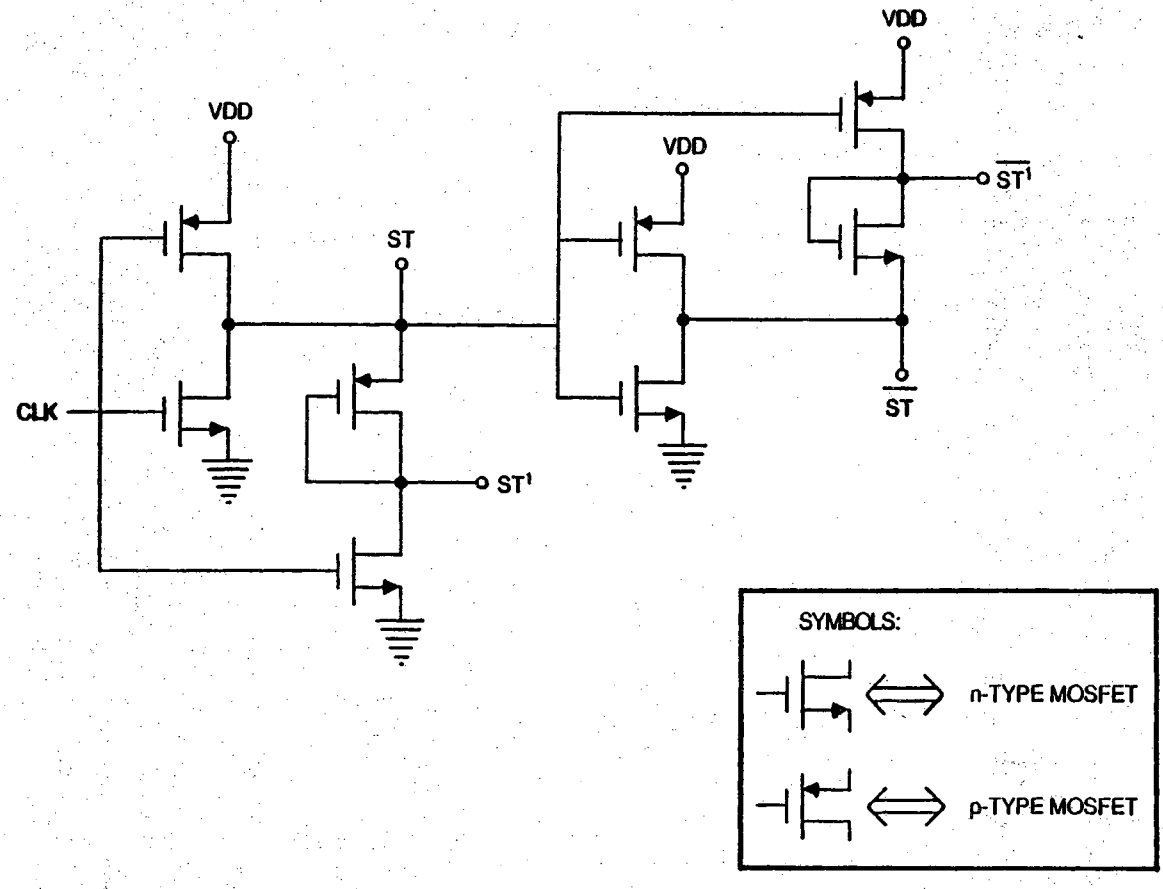


Fig. 3.8: Zipper CMOS Driver Circuit

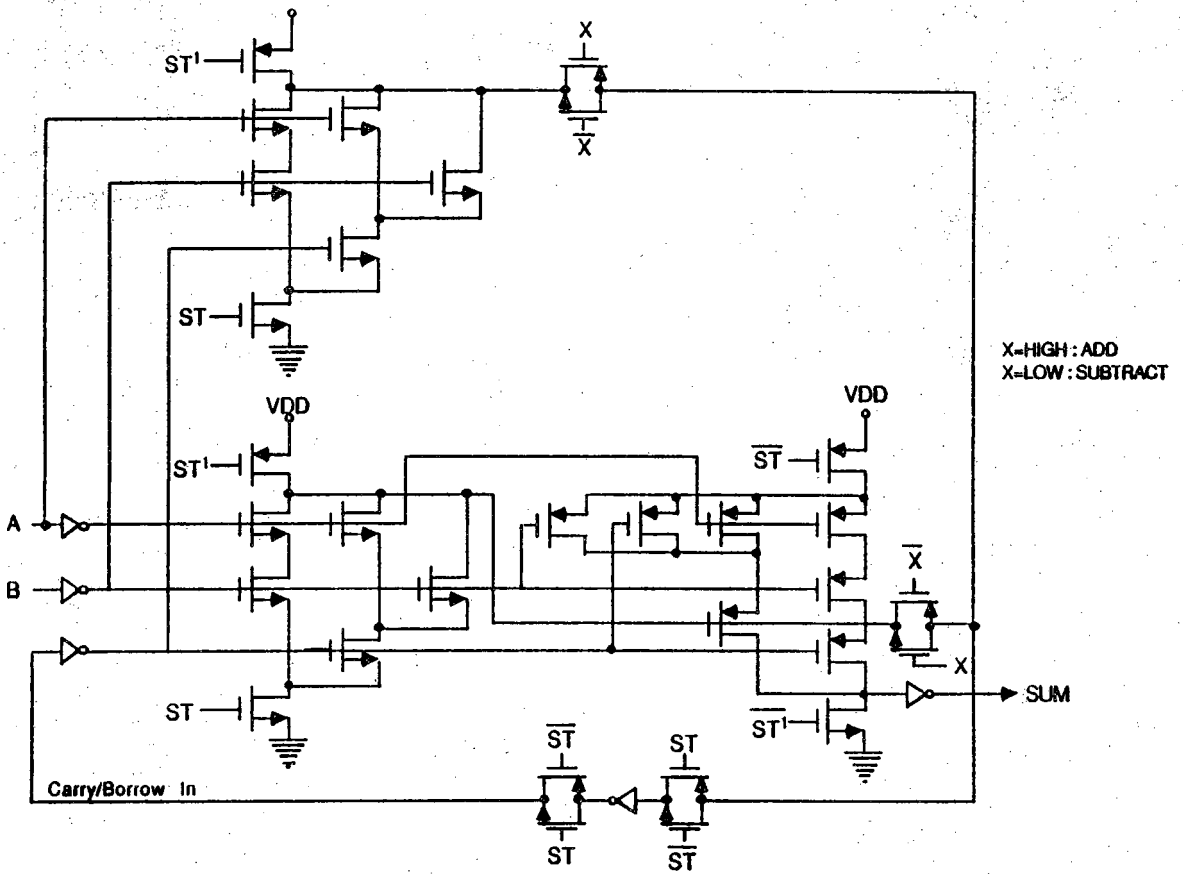


Fig. 3.9: Zipper CMOS Full Adder/Subtractor

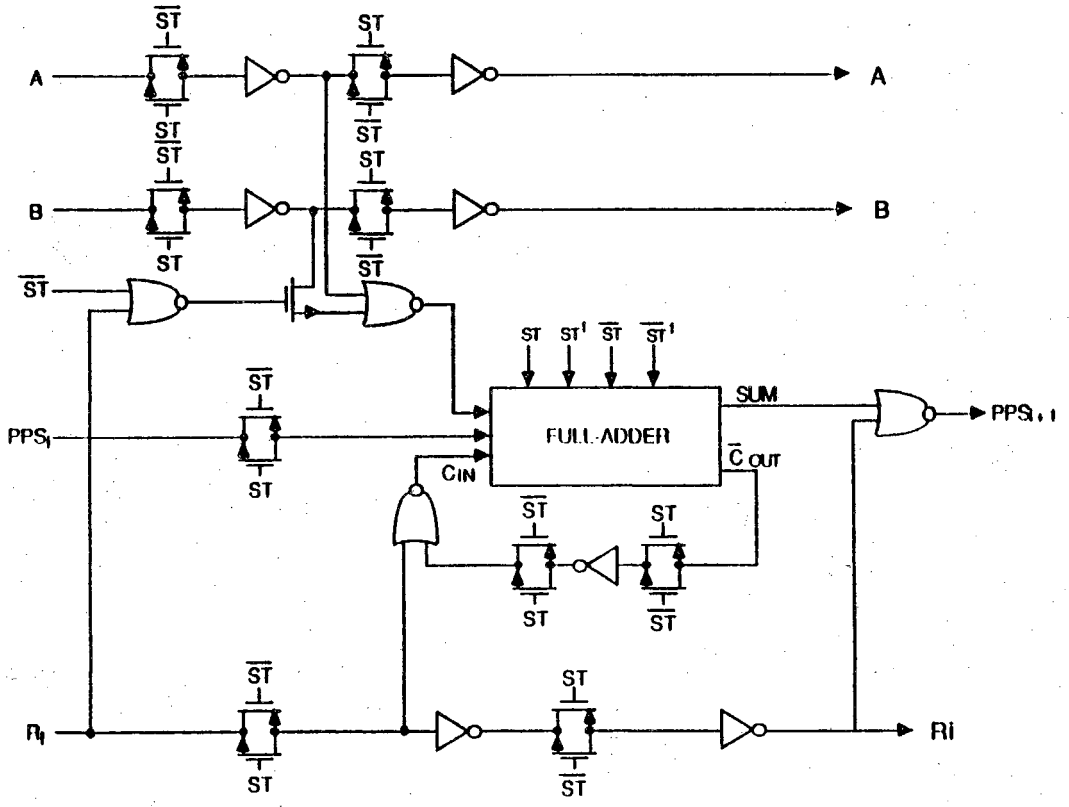


Fig. 3.10: Zipper CMOS Bit-Serial Multiplier Module

lower bound of $o(\lfloor c_1k + c_2kn \rfloor)$. A high performance multi-functional bit-serial cell architecture is described as a building block for the array configurations of the system. Zipper CMOS circuit design strategy is proposed for the cell's implementation to minimize propagation delays and maximize operating frequencies. For the case of a 16-bit system word length and 20MHz operating frequency, the total time to compute the inverse dynamics problem of a 6-link manipulator on the proposed bit-serial architecture is only 44μs.

CHAPTER 4
SYSTOLIC ARCHITECTURE FOR ROBOT INVERSE DYNAMICS COMPUTATION

4.1 Introduction

A novel systolic architecture to compute the inverse dynamics computational problem of an n-link manipulator within the systolic execution time lower bound of $o\lceil d_1 n \rceil$, where d_1 is a specified constant, is proposed. A modified form of the systolic design methodology of Moldovan and Fortes [40] is used as the primary design tool for the architecture. Modifications were required to provide allowances for unequal systolic execution times of processing cells, and to eliminate broadcasting of any input or generated variable within the systolic array. The inverse dynamics problem is decomposed into a set of directional and linear recurrence tasks amenable for direct mapping onto a fixed systolic system. This task set is composed of seven "basic" types of tasks. Thus, a basic set of systolic processors onto which these basic types of tasks may be mapped is discussed. Next, the specified systolic processor set is used as building blocks for realization of the overall systolic system. Due to alternate paths and operational delays of different systolic processors, operands may appear at muti-input modules at unequal arrival times, causing a longer pipeline time. Delay buffers must therefore be inserted to balance the pipeline. The optimal buffer assignment problem is reduced to an integer linear optimization problem. The resulting systolic architecture is thus realized in a maximally pipelined manner with a minimum number of delay buffers inserted along various computational paths.

4.2 Systolic Array Design Methodology

Various systolic array design procedures [22,23,31,40,48,59] are described in current literature. The systolic design algorithm of Moldovan and Fortes [40] is found to be the efficient since it has been proven that their design methodology produces solutions which have performance features close to those of dataflow machines. A modified

form of their systolic array design methodology will thus be used as the primary design tool for realization of systolic processors to compute the inverse dynamics problem. As mentioned earlier, some modifications to their methodology are essential to develop our particular systolic system for allowance of different computational delays of alternate systolic processors, and to eliminate broadcasting of any input or generated variable. These ideas will become more apparent by the end of section 4.6. The modified design methodology is presented below.

Procedure: SADLRE (Systolic Array Design of Linear Recurrence Equations)

Input: Algorithm A over an algebraic structure S as a 5 tuple A = (J^n,C,D,X,Y) where,
J^n is a finite index set of A, $J^n \subset I^n$;
C is a set of computations indexed by J^n;
D is a set of data dependencies;
X is a set of input variables;
Y is a set of generated variables.

Output: New dependence matrix Δ of the developed algorithm and set of tables T = {GVT, UVT, IT}, where,

GVT is a Generated Variable Table which specifies the time and cell of execution of $y_i \in Y$ at index point $\bar{j}$;

UVT is a Used Variable Table that specifies the times and cell location from which $y_i \in Y$ are available for use at index point $\bar{j}$;

IT is an Input Table that specifies the times and cell locations from which $x_i \in X$ are available for use at index point $\bar{j}$.

STEP 1: Determine a set of generated variables $\{y_{Bi}\}$ with linear indexing matrices I_{Bi} that are susceptible for broadcasting. Variable y_{Bi} is broadcasted if,

$$\text{rank}(I_{Bi}) < n$$

where n is the dimensionality of the algorithm index set J^n, and $i \in I^n$.

STEP 2: When $\{y_{Bi}\}$ has more than one variable, determine a set of indices $\{j_{Bi}\} \in J^n$ corresponding to linearly dependent columns in I_{Bi}.

STEP 3: Select a transformation Π that will provide a valid linear time scheduler, $\Pi\bar{d}_i$, $\bar{d}_i \in D$. Heuristically, Π may be found by following these steps:

(3.1) When $\{y_{Bi}\}$ has more than one variable, then $\Pi_{Bi}\bar{d}_{Bi}$ must be of the form,

$$\Pi\bar{d}_{Bi} = C_i + \sum_{j_i \in j_{Bi}} j_i C_{ji}$$

and the following conditions must be satisfied,

$$C_i \geq \sum_{j_i \in J^n / j_{Bi}} t_{ji} * S_{ji} \lfloor d_{Bi}^C \rfloor + \sum_{j_i \in j_{Bi}} t_{ji} * S_{ji} \lfloor d_{Bi}^C \rfloor \qquad (4.1)$$

$$C_{ji} \geq t_{ji} * S_{ji} \lfloor d_{Bi}^V \rfloor \quad \text{where } j_i \in j_{Bi} \qquad (4.2)$$

where, $d_{Bi} \in D$ provides data dependence vectors of variables y_{Bi}, C_i are the constant parts of $\Pi_{Bi}\bar{d}_{Bi}$, C_{ji} are the coefficients of variables $j_i \in j_{Bi}$ in $\Pi_{Bi}\bar{d}_{Bi}$, and t_{ji} is the maximum time needed for the variable y_{Bi} to travel a single unit distance in the direction of j_i. This time variable may easily be deduced from the original recurrence equation. It is equal to the interprocessor communication time, if the output y_i in the original recurrence equation is constant for any j_i. On the other hand, t_{ji} is the sum of the interprocessor communication time and cell computational delay in the case of a varying output y_i when index j_i is varied. S_{ji} represents a row of the transformation matrix S (see STEP 5) corresponding to the transformation of the j_{Bi}th index in d_{Bi}; $d_{Bi}^C \in \bar{d}_i$ is the constant part of dependence vector d_{Bi}; $d_{Bi}^V \in \bar{d}_i$ is the coefficient of the j_{Bi}th index in d_{Bi}.

(3.2) Time schedulers of input variables, $x_i \in X$, must satisfy the condition,

$$\Pi_{xi} d_{xi} \geq t_C$$

where $\Pi_{xi} \in \Pi$ transform columns $\bar{d}_{xi} \in D$ that provide data dependence vectors for input variables x_i, and t_C is the interprocessor communication time.

(3.3) Time schedulers of generated variables, $y_i \in Y$, must meet the condition,

$$\Pi_{yi}\bar{d}_{yi} \geq (t_f + t_C)$$

where $\Pi_{yi} \in \Pi$ transform columns $\bar{d}_{yi} \in D$ that provide data dependence vectors for variables y_i, and t_f is the computational delay within a cell.

(3.4) The final Π must be chosen so as to minimize the algorithm execution time,

$$t = \left\lceil \frac{\max\Pi(\bar{j}_1 - \bar{j}_2) + t_f}{\min\Pi\bar{d}_i} \right\rceil \qquad (4.3)$$

for any $\bar{j}_1, \bar{j}_2 \in J^n$, and $\bar{d}_i \in D$.

STEP 4: Select the columns of the matrix of interconnection primitives, P, as

$$P_i = d_i' \backslash d_{Bi}'$$

where $P_i \in P$, and d_i', $d_{Bi}' \in D'$; D' corresponds to the last one and two rows of dependence matrix D for one- and two-dimensional systolic arrays respectively; d_i corresponds to columns of D'; d_{Bi}' corresponds to columns of D' that provide n-1 elements of dependence vectors for variable y_{Bi}.

STEP 5: Select a second transformation S of size (n-1)×n, where n is the dimension of the algorithm index set J^n, such that:

(5.1) When $\{y_{Bi}\}$ has more than one variable, elements of S must satisfy the condition,

$$S_{jBi}\bar{d}_{Bi} = C_{i1} + j_{Bi}C_{i2}$$

where $C_{i1} \in Z^n$ and $C_{i2} \in I^n$, $C_{i2} \neq 0$ and S_{jBi} is a row of S which determines the spatial interconnect of variable y_{Bi} in the direction of j_{Bi}.

(5.2) Diophantine equation SD = PK may be solved for S, where matrix K which indicates the utilization of primitive interconnections in matrix P must satisfy the conditions of $K_{ji} \geq 0$, and $\sum_j K_{ji} \leq \Pi\bar{d}_i$.

(5.3) Matrix transformation $T = \begin{bmatrix} \Pi \\ S \end{bmatrix}$ is non-singular.

For the chosen transformation, T, the partitioning hyperplanes Π_{Pi} are given by the rows of matrix S, i.e.,

$$S = \begin{bmatrix} \Pi_{P1} \\ \Pi_{P2} \\ \vdots \\ \Pi_{P(n-1)} \end{bmatrix}$$

(5.4) Finally, S must be chosen to minimize the total number of bands, r,

$$r = \prod_{K=1}^{n-1} \left\lceil \frac{\max \Pi_{PK}(\bar{j}^1 - \bar{j}^2) + 1}{m_K} \right\rceil \tag{4.4}$$

for any $\bar{j}^1, \bar{j}^2 \in J^n$, and m_K is the width of the band.

STEP 6: Determine new dependence matrix $\Delta = TD$.

STEP 7: Determine GVT (Generated Variable Table). Significant columns of this table specify the initiation time (t_I), completion time (t_E) and location of the cell ($\bar{l}$) where index point $\bar{j}$ is processed.

$$t_I = \Pi \bar{j}$$
$$t_E = t_I + t_f$$
$$\bar{l} = S\bar{j}$$

where t_f in the computational delay of a single cell.

STEP 8: Determine UVT (Used Variable Table). Significant columns of this table specify the times $\{t_y\}$ and location of cells $\{\bar{l}_y\}$ from which the set of generated variables $y(\bar{j} - \bar{d}_y)$ are available for use at index point $\bar{j}$.

$$t_y = \Pi(\bar{j} - \bar{d}_y) + t_f$$
$$\bar{l}_y = S(\bar{j} - \bar{d}_y)$$

where d_y is the dependence vector corresponding to generated variable y.

STEP 9: Determine IT (Input Table). Significant columns of this table include the times $\{t_x\}$ and location of cells $\{\bar{l}_x\}$ from which input variables $x(\bar{j})$ are available for use at index point $\bar{j}$.

$$t_x = \Pi(\bar{j} - \bar{d}_x)$$
$$\bar{l}_x = S(\bar{j} - \bar{d}_x)$$

where $\bar{d}_x$ is the dependence vector corresponding to input variable x, and t_C

is, as defined above, the interprocessor communication time.

STEP 10: Use the new dependence matrix Δ along with the set of tables $T = \{GVT, UVT, IT\}$ to map the algorithm onto a fixed systolic architecture. The amount of time delay for buffers inserted along the paths of generated and input variables within an individual cell for synchronization purpose may be determined by,

$$t_{Dx} = \Pi\bar{d}_x - t_C$$

$$t_{Dy} = \Pi\bar{d}_y - (t_f + t_C)$$

where, t_{Dx} and t_{Dy} are the time intervals required for delay buffers inserted along the paths of an input (x) and generated variable (y) respectively. $\bar{d}_x$, $\bar{d}_y$, t_C and t_f are as defined before.

STEP 11: Partition the developed algorithm onto a set of r bands (r may be determined by equation (4.4). The mapping of the indices to processors of the partitioned algorithm is as follows: each index point j is processed within band B_i in the processor whose ith coordinate is,

$$j_{Pi} = \Pi_{Pi}\bar{j} \bmod m_i$$

where m_i is the width of the ith coordinate inside the band. It should be noted here that this step usually results in a more cost-efficient systolic architecture for algorithms with a large number of recursive iterations.

The modifications to the design methodology of [40] are included in design steps: 1, 2, 3.1-3.3, 4, 5.1, and 7-10. Step 1 determines whether any generated variable in the given linear recurrence problem may be broadcasted, which is undesired in any systolic architecture. Step 2 simply selects these variables which are amenable to broadcasting so that their respective dependence vectors may be used to construct special conditions on the selection of transformation Π in step 3.1. As described in step 3.1, Πd_{Bi} must be of the form depicted to ensure variable(s) y_{Bi} is sequentially propagated through the systolic array instead of being viable to broadcasting. Special conditions are established on the choice of C_i and C_{ji} so as to set lower bounds on propagation delays of j_{Bi} in the specified direction of data flow. The conditions on the linear time scheduler in steps 3.2 and 3.3 ensure Π is selected appropriately when the communication time, t_C, and computation time, t_f, are different. Step 4 provides a systematic method for the selection of the matrix of interconnection primitives, P, to eliminate any type of enumeration for this choice, as

delineated in [40].

Conditions set on the selection of Π alone in step 3.1 is insufficient for the elimination of broadcasting since the space scheduler S must provide the interconnections support for such data communications specified by Π. Thus, step 5.1 discusses the conditions on the choice of S to avoid broadcasts. Finally, steps 7-10 incorporate desired modifications to [40] for appropriate determination of GVT, UVT, and ΓT to the given linear recurrence problem when t_c and t_f are different.

4.3 Time Lower Bound to Compute The Inverse Dynamics Problem on a Systolic Array

The limitation on speeding up the inverse dynamics computational problem while running on a systolic array will now be discussed. Before deriving the time lower bound, the following notation must be established.

Notation:

(1) E<*l*> denotes a linear arithmetic expression of *l* distinct atoms, where an atom is a constant or variable, e.g. E<4> = a + b – c/d.

(2) T_S = Minimum time to compute the inverse dynamics problem of an n-link manipulator on a fixed systolic array.

Theorem 4.1: The minimum time to compute the inverse dynamics problem of an n-link manipulator on a fixed systolic architecture is bounded below by $o\lceil d_1 n \rceil$, where d_1 is a specified constant.

Proof: Using equation (4.3) in procedure SADLRE, the systolic processing time, t, for any task in program Newton-Euler may be expressed in the general form,

$$t = \left\lceil \max\Pi \frac{\left[\begin{bmatrix} n \\ \max a_2 \\ \vdots \\ \max a_m \end{bmatrix} - \begin{bmatrix} 1 \\ \min a_2 \\ \vdots \\ \min a_m \end{bmatrix}\right] + t_f}{\min \Pi \bar{d}_i} \right\rceil$$

$$= \left\lceil \frac{\max \Pi_1(n-1) + \sum_{i=2}^{m} \max \Pi_i(\max a_i - \min a_i) + t_f}{\min \Pi \bar{d}_i} \right\rceil$$

where n is the number of manipulator joints, $\max [a_2,\ldots,a_m]$ and $\min [a_2,\ldots,a_m]$ are upper and lower bounds of the task's index set $\{\bar{j}_2,\ldots,\bar{j}_m\}$, respectively. Variables $\sum_{i=2}^{m} \max \Pi_i (\max a_i - \min a_i)$, t_f, and $\min \Pi d_i$ in the above equation are constants independent of n. In the case of $\Pi_1 = f(j_o)$ where Π_1 is the first term in the vector transformation and j_o is the outermost index of the iterative algorithm, $\max \Pi_1(n-1)$ will be $o(n^2)$. Thus, for minimal t, $\Pi_1 \neq f(j_o)$. Under this condition for Π to achieve the lower bound for t, we may rewrite the above above formulation of t as,

$$t = \left\lceil \frac{b_1(n-1) + b_2}{b_3} \right\rceil$$

$$= c_1 n + c_2$$

where b_1, b_2, b_3, c_1, and c_2 are specified constants. Thus, the lower bound to evaluate T_S for an arithmetic expression E<n> may be defined as,

$$T_S[E<n>] \geq o(c_1 \lceil n \rceil)$$

The inverse dynamics problem may be considered as computing a set of arithmetic operations which result in obtaining the joint torques. Each joint torque τ_i of joint i can be expressed as an arithmetic expression containing at least 3n atoms: n joint positions $\{q_i\}_{i=1}^{n}$, n joint velocities $\{\dot{q}_i\}_{i=1}^{n}$ and n joint accelerations $\{\ddot{q}_i\}_{i=1}^{n}$, implying,

$$T_S[E<3n>] \geq o(3c_1 \lceil n \rceil) \qquad (4.5)$$

Rewriting equation (4.5) without changing the order of the lower bound to evaluate a set of n torques,

$$T_S[\tau_1,\ldots,\tau_n] \geq o(d_1\lceil n\rceil)$$

where d_1 is a specified constant. Q.E.D.

By establishing a lower bound in Theorem 4.1, alternate systolic architectures for the computation of the inverse dynamics problem may be compared and contrasted by simply comparing coefficient d_1 of the competing systolic structures.

4.4 Reformulation of Newton Euler Equations of Motion for Direct Mapping onto a Fixed Systolic Architecture

In this section we shall reconstruct the Newton-Euler dynamics algorithm in Appendix A in a form amenable for direct mapping onto a fixed systolic array. This reformulation is achieved by decomposing the algorithm into a set of unidirectional and linear recurrence computational tasks. The algorithm is as follows:

Algorithm NEWTON_EULER

Forward Iterations:

```
for h = 0 to (n−1) do
  begin h
    for i = 0 to 2 do
      begin i
        For j = 0 to 2 do
          For k = 0 to 1 do
    Start Task 1
        begin j
          ω[h,i,j] = ω[h,i,j−1] + R[i,j](ω[h−1,i,2] + θ̇z[h,i])
        end j
    Finish Task 1
    Start Task 2
        if i = 2 then
          x1[h,i] = ω[h,2]*θ̇ z[h,0] − ω[h,0]*θ̇ z[h,2]
        else
          x1[h,i] = ω[h,i]*θ̇ z[h,i+1] − ω[h,i+1]*θ̇ z[h,i]
    Finish Task 2
```

Start Task 3

$x_2[h,i] = x_1[h,i] + \ddot{\theta}^z[h,i]$

Finish Task 3

Start Task 4

begin j

$\dot{\omega}[h,i,j] = \dot{\omega}[h,i,j-1] + R[i,j](\dot{\omega}[h-1,i,2] + x_2[h,i])$

end j

Finish Task 4

Start Task 5

if i = 2 then

$x_3[h,i] = \dot{\omega}[h,2]*P[h,0] - \dot{\omega}[h,0]*P[h,2]$

else

$x_3[h,i] = \dot{\omega}[h,i]*P[h,i+1] - \dot{\omega}[h,i+1]*P[h,i]$

Finish Task 5

Start Task 6

begin k

$x_4[h,i,k=0] = \omega[h,i]$

if i = 2 then

$x_4[h,i,k] = x_4[h,2,k-1]*P[h,0,k] - x_4[h,0,k-1]*P[h,2,k]$

else

$x_4[h,i,k] = x_4[h,i,k-1]*P[h,i+1,k] - x_4[h,i+1,k-1]*P[h,i,k]$

end k

Finish Task 6

Start Task 7

$x_5[h,i] = x_3[h,i] + x_4[h,i]$

Finish Task 7

Start Task 8

begin j

$\dot{v}[h,i,j] = \dot{v}[h,i,j-1] + R[i,j]*\dot{v}[h-1,i,2] + x_5[h,i]$

end j

Finish Task 8

Start Task 9

if i = 2 then

$x_6[h,i] = \dot{\omega}[h,2]*P_C[h,0] - \dot{\omega}[h,0]*P_C[h,2]$

else

$x_6[h,i] = \dot{\omega}[h,i]*P_C[h,i+1] - \dot{\omega}[h,i+1]*P_C[h,i]$

Finish Task 9

Start Task 10
  begin k
    $x_7[h,i,k=0] = \omega[h,i]$
  if i = 2 then
    $x_7[h,i,k] = x_7[h,2,k-1]*P_C[h,0,k] - x_7[h,0,k-1]*P_C[h,2,k]$
  else
    $x_7[h,i,k] = x_7[h,i,k-1]*P_C[h,i+1,k] - x_7[h,i+1,k-1]*P_C[h,i,k]$
  end k
Finish Task 10
Start Task 11
    $x_8[h,i] = x_6[h,i] + x_7[h,i]$
Finish Task 11
Start Task 12
    $\dot{v}_C[h,i] = \dot{v}[h,i] + x_8[h,i]$
Finish Task 12
Start Task 13
    $F[h,i] = m[h]*\dot{v}_C[h,i]$
Finish Task 13
Start Task 14
  begin j
    $x_9[h,i,j] = x_9[h,i,j-1] + I[i,j]*\omega[h,i]$
  end j
Finish Task 14
Start Task 15
  if i = 2 then
    $x_{10}[h,i] = \omega[h,2]*x_9[h,0] - \omega[h,0]*x_9[h,2]$
  else
    $x_{10}[h,i] = \omega[h,i]*x_9[h,i+1] - \omega[h,i+1]*x_9[h,i]$
Finish Task 15
Start Task 16
  begin j
    $x_{11}[h,i,j] = x_{11}[h,i,j-1] + I[i,j]*\dot{\omega}[h,i]$
  end j
Finish Task 16
Start Task 17
    $N[h,i] = x_{10}[h,i] + x_{11}[h,i]$
Finish Task 17

```
    end i
end h
```

Backward Iterations:

```
for h = n to 1 do
  begin h
    for i = 0 to 2 do
      begin i
        for j = 0 to 2 do
    Start Task 18
      begin j
        f[h,i,j] = f[h,i,j−1] + R[i,j]*f[h+1,i] + F[h,i]
      end j
    Finish Task 18
    Start Task 19
        x12[h,i] = P[h,i] + PC[h,i]
    Finish Task 19
    Start Task 20
      if i = 2 then
        x13[h,i] = x12[h,2]*F[h,0] − x12[h,0]*F[h,2]
      else
        x13[h,i] = x12[h,i]*F[h,i+1] − x12[h,i+1]*F[h,i]
    Finish Task 20
    Start Task 21
        x14[h,i] = N[h,i] + x13[h,i]
    Finish Task 21
    Start Task 22
      if i = 2 then
        x15[h,i] = P[h,2]*f[h+1,0] − P[h,0]*f[h+1,2]
      else
        x15[h,i] = P[h,i]*f[h+1,i+1] − P[h,i+1]*f[h+1,i]
    Finish Task 22
    Start Task 23
      begin j
        n[h,i,j] = n[h,i,j−1] + R[i,j](n[h+1,i,2] + x15[h,i]) + x14[h,i]
      end j
```

```
    Finish Task 23
    Start Task 24
        τ[h,i] = τ[h,i−1] + R^z[h,i] * n[h,i]
    Finish Task 24
  end i
end h
```

4.5 Systematic Design of a Basic Set of Systolic Processors for the Inverse Dynamics Computational Problem

From the algorithm of Section 4.4, it is seen that a ''basic'' set of types of tasks exist from which the complete systolic architecture for the computation of the inverse dynamics problem may be derived. Thus, by designing a set of systolic processors onto which this ''basic'' set of task types may be mapped, we can arrive at the complete systolic architecture for the computation of the specified problem by simply using these processors as building blocks for the realization of the overall systolic system. The basic set of tasks may be partitioned into a set of unidirectional and linear recurrence tasks as follows:

Unidirectional tasks:

Type 1: Vector-Add (VA)

```
for i = 0 to 2 do
   y[i] = x[i] + z [i]
end i
```

Type 2: Scalar-Vector Product (SVP)

```
for i = 0 to 2 do
   y[i]=K*x[i]
end i
```

Type 3: Vector Cross-Product (VCP)

```
for i = 0 to 2 do
   if i = 2 then
      y[i] = x[2] * z[0] − x[0] * z[2]
   else
      y[i] = x[i] * z[i+1] − x[i+1] * z[i]
end i
```

Linear Recurrence tasks:

Type 4: Inner Product (IP)

```
for i = 0 to 2 do
    y[i] = y[i−1] + x[i] * z[i]
end i
```

Type 5: Matrix-Vector Product

```
for i = 0 to 2 do
   for j = 0 to 2 do
      y[i,j] = y[i,j−1] + A[i,j] * x[i]
   end j
end i
```

Type 6: Recursive Matrix-Vector Product (RMVP)

Type 6a:

```
   for h = 0 to n−1 do
      for i = 0 to 2 do
         for j = 0 to 2 do
            y[h,i,j] = y[h,i,j−1] + A[i,j] (y[h−1,i,2] + x[h,i])     (1)
         end j
      end i
   end h
```

Type 6b:

Replace line (1) in Type 6a by,

```
y[h,i,j] = y[h,i,j−1] + A[i,j] * y[h−1,i,2] + x[h,i]
```

Type 6c:

Replace line (1) in Type 6a by,

```
y[h,i,j] = y[h,i,j−1] + A[i,j] * (y[h−1,i,2] + x[h,i]) + z[h,i]
```

Type 7: Recursive Vector Cross-Product (RVCP)

```
for i = 0 to n
   for j = 0 to 2 do
      if j = 2 then
         y[h,i] = y[i−1,2] * x[i,0] − y[i−1,0] * x[i,2]
      else
         y[h,i] = y[i−1,j] * x[i,j+1] − y[i−1,j+1] * x[i,j]
   end j
end i
```

Now that the basic types of tasks prevalent in the inverse dynamics computational problem have been defined, we can develop systolic architectures onto which these processes may be mapped. The modified systolic mapping procedure of Section 4.2 will be used to develop the systolic architectures for linear recurrence tasks 4-7.

For the purpose of generality while deriving systolic architectures using procedure SADLRE, we shall assume that the interprocessor communication time, t_c, is of unit time latency and the operational delay of primitive tasks such as addition, subtraction and multiplication is also an unit of time.

Before deriving the specified systolic processor architectures, two terms which will be used hereafter must be defined. The first one is, **Systolic Operational Delay,** which is defined as the latency required to compute the first element of a vector operation at the end of the first iteration in the given linear recurrence problem on the specified systolic array. Next, the **Systolic Execution Time** is defined as the latency to evaluate all elements of the vector operation at the end of n iterations in the linear recurrence problem on the specified systolic array.

4.5.1 Systolic Processors for Unidirectional Tasks

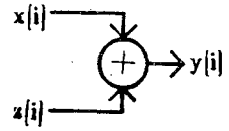
Task types 1 and 2 are represented by primitive adder and multipler cells (Figs. 4.1(a), (b)). Paired vector operands enter these cells in synchronous fashion. A systolic vector cross-product module (type 3 task) is illustrated in Fig. 4.1(c). Unit delays are inserted along some input variable paths for operation synchronization. The operation delay of this particular module is 3 time units.

4.5.2 Systolic Architecture for a Type 4 Task

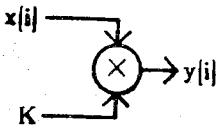
The Algol-like program for an inner product operation is,

```
for i = 0 to 2 do
    j = i
    x[i,j] = x[i,j−1]
    z[i,j] = z[i,j−1]
    y[i] = y[i-1] + x[i,j] * z[i,j]
end i
```

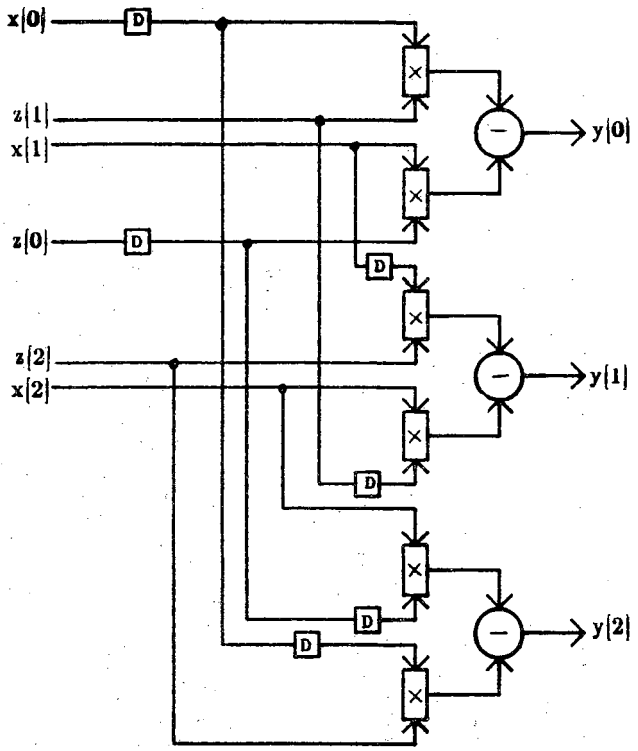
The data dependency matrix for the above algorithm is $D = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$, where the columns from left to right denote data dependencies for x[i], z[i] and y[i] respectively.



(a): Type 1 Processor



(b): Type 2 Processor



(c): Type 3 Processor

Fig. 4.1: Systolic Processors for Unidirectional Tasks

By following procedure SADLRE, Π is found to be [3 1], and S to be $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. The new transformation matrix, Δ, is thus $\begin{bmatrix} 1 & 1 & 3 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$. Tables 4.1-4.3 show the GVT, UVT, and IT for this type of task. The systolic architecture and its partitioned structure may be derived from these tables to be as shown in Fig. 4.2(a) and (b) respectively. Also, the architecture of an individual processing element of Fig. 4.2(a) is illustrated in Fig. 4.2(c). Information regarding scheduling of operands and generation of intermediate variables in Fig. 4.2(a) can be obtained from task type 4's tables {T}. The operational delay for the Type 4 task is eight time units.

4.5.3 Systolic Architecture for a Type 5 Task

The Algol-like program of a matrix-vector product (Task 4) operation will now be presented in a slightly modified form,

```
for i = 0 to 2 do
    for j = 0 to 2 do
        x[i] = x[i−1]                              (1)
        A[i,j] = A[i,j−1]                          (2)
        y[i,j] = y[i,j−1] + A[i,j] * x[i]
    end j
end i
```

The purpose of statements (1) and (2) is to ensure the inputs are propagated sequentially along the ith and jth directions, respectively, of the systolic array. These statements, therefore, eliminate any possibility of broadcasting input variables. The data dependency matrix, D, of iteration (i,j) for the above algorithm may easily be derived as,

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

where columns 1, 2 and 3 represent dependence vectors for variables x[i−1], A[i,j−1] and y[i,j−1] respectively.

Linear indexing matrices, I_{Bi}, for this problem were found to have the same rank as the dimensionality of the algorithm index set. Thus, it may be assumed that broadcasting of any input or generated variable will not occur. We can, therefore, jump to step 3.2 in procedure SADLRE.

Table 4.1: GVT of a Type 4 Task

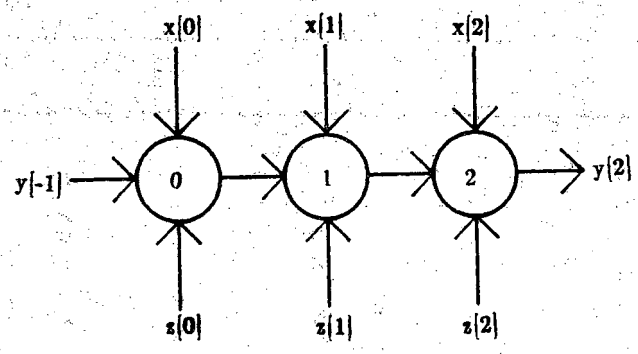
ITERATION (i)	BEGIN EXECUTION AT TIME	FINISH EXECUTION AT TIME	IN CELL (i)	GENERATING VARIABLE
0	0	2	0	y(0)
1	3	5	1	y(1)
2	6	8	2	y(2)

Table 4.2: UVT of a Type 4 Task

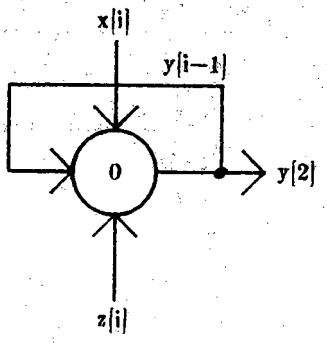
ITERATION (i)	USING VARIABLES	FROM CELL $(i - \bar{d}_y)$	AT TIMES
0	y(-1)	-1	-1
1	y(0)	0	2
2	y(1)	1	5

Table 4.3: IT of a Type 4 Task

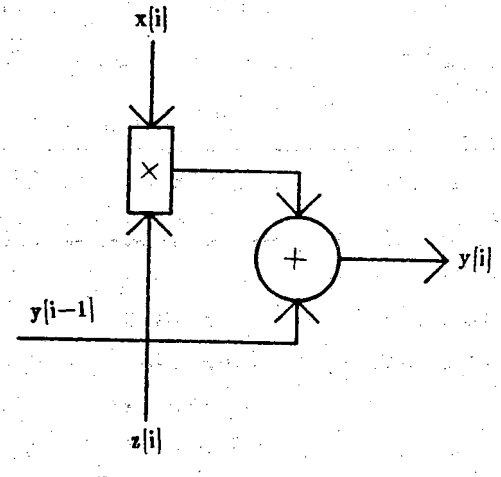
ITERATION (i)	USING INPUTS x(i), z(i)	FROM CELL $(i,j - \bar{d}_x)$	AT TIMES
0	x(0), z(0)	0,-1	-1
1	x(1), z(1)	1,-1	2
2	x(2), z(2)	2,-1	5



(a): Systolic Architecture



(b): Partitioned Systolic Architecture



(c): An Individual Processing Element

Fig. 4.2: Systolic Architecture for a Type 4 Task

From the above Algol-like program specification, it is easily derived that the computational delay of an individual systolic cell will be two time units since two consecutive primitive operations are required to evaluate y[i,j]. The linear time scheduler, ΠD, must, therefore satisfy the conditions of $\Pi\bar{d}_1, \Pi\bar{d}_2 \geq 1$ and $\Pi\bar{d}_3 \geq 3$, where the subscripts of dependence vector d indicate column indexing in matrix D. The transformation vector, Π, which meets the above specifications and minimizes the algorithm execution time, t, in step 3.4, is found to be [1 3].

Using step 4, we may obtain a two-dimensional systolic array architecture by selecting the matrix of interconnection primitives, P, as $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Further, the second transformation S is chosen to allow the diophantine equation SD = PK to be solvable for S, and minimize the number of bands, r. The derived S-transformation and interconnection utilization matrix, K, are as follows:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ for } K = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

We thus arrive at the new dependence matrix, Δ,

$$\Delta = TD = \begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 3 & 3 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

The first row of the transformed matrix, Δ, indicates the number of time units allowed for the respective variable to travel from the processor where it is generated to the processor where it is used. The next two rows specify the direction of data flow for the respective variable. It is therefore easily verified that input x[i] needs one time unit to propagate one space in the ith direction, input A[i,j] and generated variable y[i,j−1] requires three time units to travel one space in the jth direction.

Tables 4.4 through 4.6 delineate the GVT, UVT and IT, respectively, of the matrix-vector product operation. They are derived using Steps 7 through 9 of procedure SADLRE. The systolic array and its partitioned form inferred from these tables are illustrated in Fig. 4.3(a) and (b) respectively. The architecture for an individual cell of the systolic architecture using step 10 is shown in Fig. 4.3(c). The operational delay of the systolic module is 8 time units to perform the designated

Table 4.4: GVT of a Type 5 Task

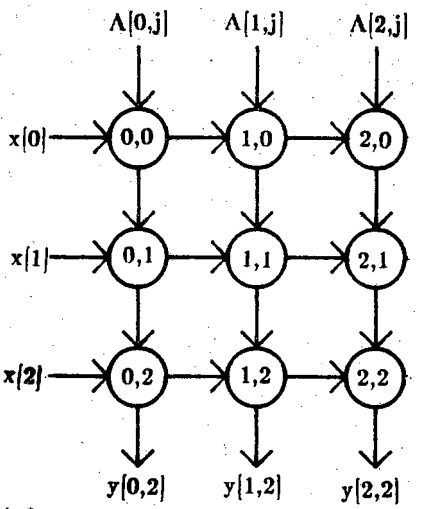
ITERATION (i,j)	BEGIN EXECUTION AT TIME	FINISH EXECUTION AT TIME	IN CELL (i,j)	GENERATING VARIABLE
0,0	0	2	(0,0)	y(0,0)
0,1	3	5	(0,1)	y(0,1)
0,2	6	8	(0,2)	y(0,2)
1,0	1	1	(1,0)	y(1,0)
1,1	4	6	(1,1)	y(1,1)
1,2	7	9	(1,2)	y(1,2)
2,0	2	4	(2,0)	y(2,0)
2,1	5	7	(2,1)	y(2,1)
2,2	8	10	(2,2)	y(2,2)

Table 4.5: UVT of a Type 5 Task

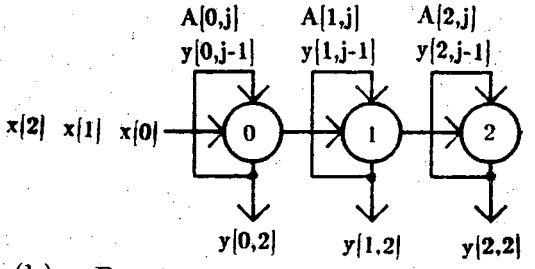
ITERATION (i,j)	USING VARIABLE	FROM CELL	AT TIME
0,0	(0,-1)	(0,-1)	-3
0,1	(0,0)	(0,0)	0
0,2	(0,1)	(0,1)	3
1,0	(1,-1)	(1,-1)	-2
1,1	(1,0)	(1,0)	1
1,2	(1,1)	(1,1)	4
2,0	(2,-1)	(2,-1)	-1
2,1	(2,0)	(2,0)	2
2,2	(2,1)	(2,1)	5

Table 4.6: IT of a Type 5 Task

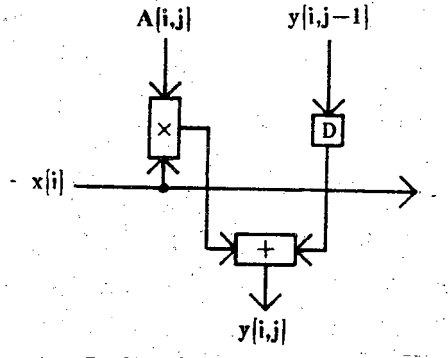
ITERATION (i,j)	USING INPUTS	FROM CELLS	AT TIMES
0,0	x[0], A[0,0]	(-1,0), (0,-1)	-1,-3
0,1	x[0], A[0,1]	(-1,1), (0,0)	2,0
0,2	x[0], A[0,2]	(-1,2), (0,1)	5,3
1,0	x[1], A[1,0]	(0,0), (1,-1)	0,-2
1,1	x[1], A[1,1]	(0,1), (1,0)	3,1
1,2	x[1], A[1,2]	(0,2), (1,1)	6,4
2,0	x[2], A[2,0]	(1,0), (2,-1)	1,-1
2,1	x[2], A[2,1]	(1,1), (2,0)	4,2
2,2	x[2], A[2,2]	(1,2), (2,1)	7,5



(a): Systolic Architecture



(b): Partitioned Systolic Architecture



(c): An Individual Processing Element

Fig. 4.3: Systolic Architecture for a Type 5 **Task**

computation.

4.5.4 Systolic Architecture for Task Types 6(a)-(c)

Type 6(a) Task

The recursive matrix-vector product formulation of a Type 6(a) task is represented by the following modified Algol-like program:

```
for h = 0 to n−1 do
  for i = 0 to 2 do
    for j = 0 to 2 do
      x[h,i] = x[h,i−1]
      A[i,j] = A[i,j−1]
      y[h,i,j] = y[h,i,j−1] + A[i,j] (y[h−1,i,2] + x[h,i])
    end j
  end i
end h
```

The data dependency matrix for the above algorithm specification may be derived to be,

$$D = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & j-2 \end{bmatrix}$$

where the columns from left to right specify data dependence vectors for variables x[h,i-1], A[i,j-1], y[h,i,j-1] and y[h−1,i,2] respectively.

The linear indexing matrix for the generated variable y[h−1,i,2] is,

$$I_{By[h-1,i,2]} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} .$$

Since the rank of the above linear indexing matrix is less than the dimensionality of the algorithm index set, it is therefore clear that the generated variable y[h−1,i,2] may be broadcasted. By using Step 2 and 3.1 of procedure SADLRE, the linear time scheduler for variable y[h-1,i,2] must be of the form,

$$\Pi \bar{d}_{By[h-1,i,2]} = C_1 + jC_2 \qquad (4.6)$$

The conditions on constants, C_1 and C_2, may only be determined after the selection of the S transformation matrix. By Step 4, we find the matrix of interconnection

primitives, P, to be,

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

Using Step 5.1, broadcasting of variable y[h−1,i,2] is avoided by also setting the following condition on the space scheduler transformation matrix S,

$$S_{By[h-1,i,2]}\bar{d}_{By[h-1,i,2]} = C_3 + jC_4 \tag{4.7}$$

where $C_3 \in Z^n$, $C_4 \in I^n$ and $C_4 \neq 0$. A S-matrix is thus selected which minimizes the number of bands, r, satisfies equation (4.7), and allows the diophantine equation SD = PK to be solvable for S,

$$S = \begin{bmatrix} 0 & 1 & 0 \\ 3 & 0 & 1 \end{bmatrix} \text{ for } K = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

From this choice of S, constants C_3 and C_4 in Eq. (4.7) are both one.

We may now return to the problem of setting conditions on C_1 and C_2 in Eq. (4.6). By Eqs. (4.6) and (4.7) in Step 3, C_1 and C_2 must both be greater or equal to 4 since $t_j \geq t_{fj} + t_C(=4)$ and $S_{y[h-1,i,2]}\, d^C_{By[h-1,i,2]}$, $S_{y[h-1,i,2]}\, d^V_{By[h-1,i,2]}$ are both one. Using Steps 3.2 and 3.3, we impose further conditions on the selection of $\Pi : \Pi d_{x[h,i-1]}$, $\Pi d_{A[i,j-1]} \geq 1$, and $\Pi \bar{d}_{y[h,i,j-1]} \geq 4$. Finally, the transformation Π is chosen to satisfy all the above conditions and minimize the algorithm execution time t, $\Pi = [12\ \ 1\ \ 4]$.

The new transformation matrix, Δ, is thus,

$$\Delta = TD = \begin{bmatrix} 1 & 4 & 4 & 4+4j \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1+j \end{bmatrix}$$

The information gathered from this Δ matrix regarding communication latency for the directional flow of variables in the systolic array is as follows:

(1) x[h,i−1] requires 1 time unit to travel 1 space unit in the ith direction.

(2) A[i,j−1] needs 4 units of time to move 1 space unit in the jth direction.

(3) Generated variable, y[h,i,j−1], travels 1 space unit in the jth direction in 4 time units.

(4) Generated variable, y[h−1,i,2], requires 4+4j units of time to travel 1+j space units in the jth direction.

The set of tables {GVT, UVT, IT} for the Type 6a operation is depicted in Tables 4.7 through 4.9, respectively. The systolic array architecture derived from these tables is shown in Fig. 4.4(a). The number of systolic cells required for this architecture is 3n, where n is the number of manipulator joints, and the operational delay of the array is 11 units of time.

The architecture of an individual cell in the systolic array of Fig. 4.4(a) is illustrated in Fig. 4.4(c). Note that buffer delays derived using Step 10 of SADLRE generated insufficient latencies along the paths of A[i,j] and y[h,i,j−1] due to the asynchronous sequence in which these variables are operated upon in the cell. This problem is solved by adding extra latencies along their paths for the purpose of operation synchronization.

The algorithm may be mapped onto a more cost-efficient architecture of a 3×3 VLSI array by following the partitioning technique in Step 11. This partitioned architecture onto a set of 3 bands is illustrated in Fig. 4.4(b). The scheduling of these bands is performed by assigning the processing of $y(\bar{j})$ in band B_i, $1 \le i \le 3$, at a processor whose ith coordinate is $\Pi_{Pi}\bar{j} \bmod 3$, where Π_{Pi} corresponds to the ith row in the S matrix.

Type 6(b) Task

The systolic architecture for this type of task is identical to that of a Type 6(a) task. The only architectural modification exists within an individual processing cell. Fig. 4.5 specifies the processing cell architecture for a Type 6(b) task. The shorter computational latency of this cell compared to that of a Type 6(a) task, thus, requires the systolic array for a Type 6(b) task to have a shorter operation delay of 8 time units.

Type 6(c) Task

The systolic architecture of this task type is also the same as that of a Type 6(a) task. An individual processing cell architecture for a Type 6(c) task is shown in Fig. 4.6. The operational delay for the type type 6(c) task is indifferent from that of a Type 6(a) task, i.e. a latency of 11 units.

4.5.5 Systolic Array for a Type 7 Task

The Algol-like program for a type 7 recursive vector cross-product operation is as follows,

```
for i = 0 to n−1 do
    for j = 0 to 2 do
```

Table 4.7: GVT of a Type 6a Task

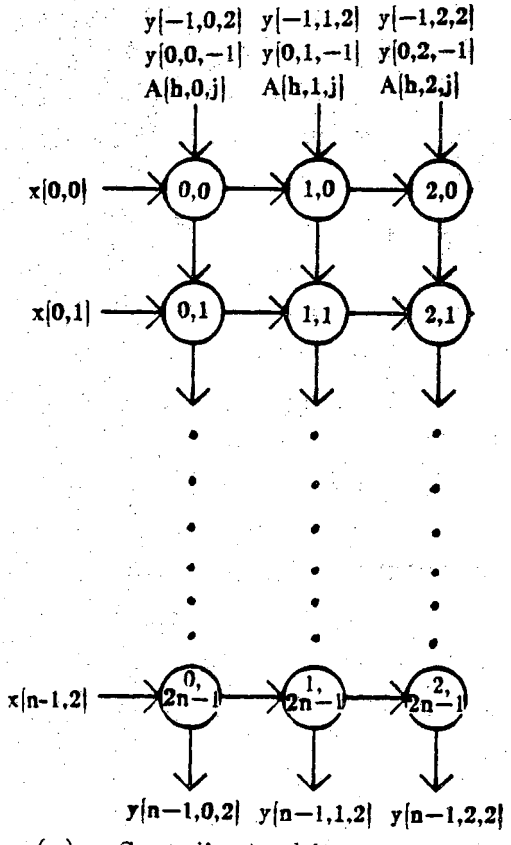
ITERATION (h,i,j)	BEGIN EXECUTION AT TIME	FINISH EXECUTION AT TIME	IN CELL (i,j)	GENERATING VARIABLE
0,0,0	0	3	0,0	y(0,0,0)
0,0,1	4	7	0,1	y(0,0,1)
0,0,2	8	11	0,2	y(0,0,2)
0,1,0	1	4	1,0	y(0,1,0)
0,1,1	5	8	1,1	y(0,1,1)
0,1,2	9	12	1,2	y(0,1,2)
0,2,0	2	5	2,0	y(0,2,0)
0,2,1	6	9	2,1	y(0,2,1)
0,2,2	10	13	2,2	y(0,2,2)
1,0,0	12	15	0,3	y(1,0,0)
1,0,1	16	19	0,4	y(1,0,1)
1,0,2	20	23	0,5	y(1,0,2)
1,1,0	13	16	1,3	y(1,1,0)
1,1,1	17	20	1,4	y(1,1,1)
1,1,2	21	24	1,5	y(1,1,2)
1,2,0	14	17	2,3	y(1,2,0)
1,2,1	18	21	2,4	y(1,2,1)
1,2,2	22	25	2,5	y(1,2,2)

Table 4.8: UVT of a Type 6a Task

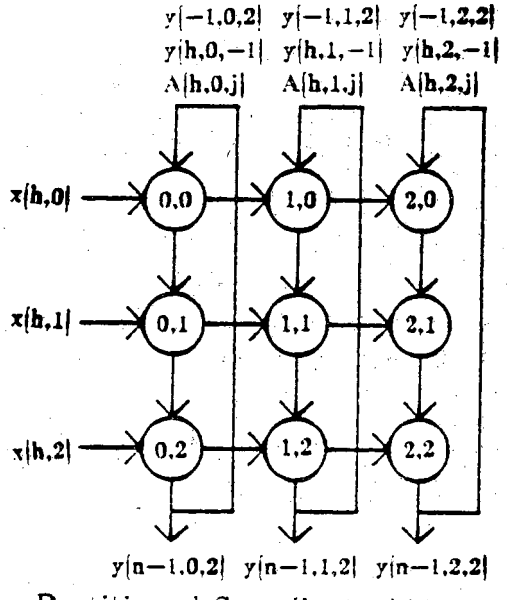
ITERATION (h,i,j)	USING VARIABLES	FROM CELLS	AT TIMES
0,0,0	y(0,0,-1), y(-1,0,2)	(0,1), (0,-1)	-1,-1
0,0,1	y(0,0,0), y(-1,0,2)	(0,0), (0,-1)	3,-1
0,0,2	y(0,0,1), y(-1,0,2)	(0,1), (0,-1)	7,-1
0,1,0	y(0,1,-1), y(-1,1,2)	(1,-1), (1,-1)	0,0
0,1,1	y(0,1,0), y(-1,1,2)	(1,0) (1,-1)	4,0
0,1,2	y(0,1,1), y(-1,1,2)	(1,1), (1,-1)	8,0
0,2,0	y(0,2,-1), y(-1,2,2)	(2,-1), (2,-1)	1,1
0,2,1	y(0,2,0), y(-1,2,2)	(2,0), (2,-1)	5,1
0,2,2	y(0,2,1), y(-1,2,2)	(2,1), (2,-1)	9,1
1,0,0	y(1,0,-1), y(0,0,2)	(0,2), (0,2)	11,11
1,0,1	y(1,0,0), y(0,0,2)	(0,3), (0,2)	15,11
1,0,2	y(1,0,1), y(0,0,2)	(0,4), (0,2)	19,11
1,1,0	y(1,1,-1), y(0,1,2)	(1,2), (1,2)	12,12
1,1,1	y(1,1,0), y(0,1,2)	(1,3), (1,2)	16,12
1,1,2	y(1,1,1), y(0,1,2)	(1,4), (1,2)	20,12
1,2,0	y(1,2,-1), y(0,2,2)	(2,2), (2,2)	13,13
1,2,1	y(1,2,0), y(0,2,2)	(2,3), (2,2)	17,13
1,2,2	y(1,2,1), y(0,2,2)	(2,4), (2,2)	21,13

Table 4.9: IT of a Type 6a Task

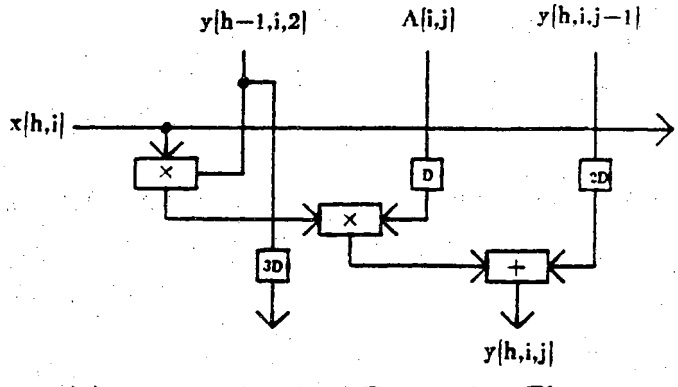
ITERATION (h,i,j)	USING INPUT x(h,j), A(i,j)	FROM CELLS	AT TIMES
0,0,0	x(0,0), A(0,0)	(-1,0), (0,-1)	-1,-4
0,0,1	x(0,1), A(0,1)	(-1,1), (0,0)	3,0
0,0,2	x(0,2), A(0,2)	(-1,2), (0,1)	7,4
0,1,0	x(0,0), A(1,0)	(0,0), (1,-1)	0,-3
0,1,1	x(0,1), A(1,1)	(0,1), (1,0)	4,1
0,1,2	x(0,2), A(1,2)	(0,2), (1,1)	8,5
0,2,0	x(0,0), A(2,0)	(1,0), ((2,-1)	1,-2
0,2,1	x(0,1), A(2,1)	(1,1,), (2,0)	5,2
0,2,2	x(0,2), A(2,2)	(1,2), (2,1)	9,6
1,0,0	x(1,0), A(0,0)	(-1,3), (0,2)	11,8
1,0,1	x(1,1), A(0,1)	(-1,4), (0,3)	15,12
1,0,2	x(1,2), A(0,2)	(-1,5), (0,4)	19,16
1,1,0	x(1,0), A(1,0)	(0,3), (1,2)	12,9
1,1,1	x(1,1), A(1,1)	(0,4), (1,3)	16,13
1,1,2	x(1,2), A(1,2)	(0,5), (1,4)	20,17
1,2,0	x(1,0), A(2,0)	(1,3), (2,2)	13,10
1,2,1	x(1,1), A(2,1)	(1,4), (2,3)	17,14
1,2,2	x(1,2), A(2,2)	(1,5), (2,4)	21,18



(a): Systolic Architecture



(b): Partitioned Systolic Architecture



(c): An Individual Processing Element

Fig. 4.4: Systolic Architecture for a Type 6a Task

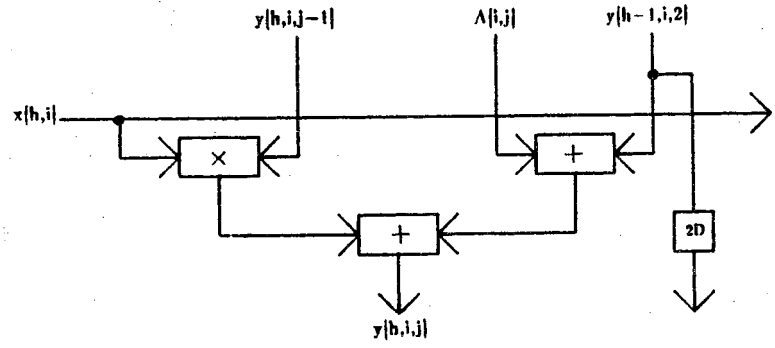


Fig. 4.5: Processing Element for a Type 6b Task

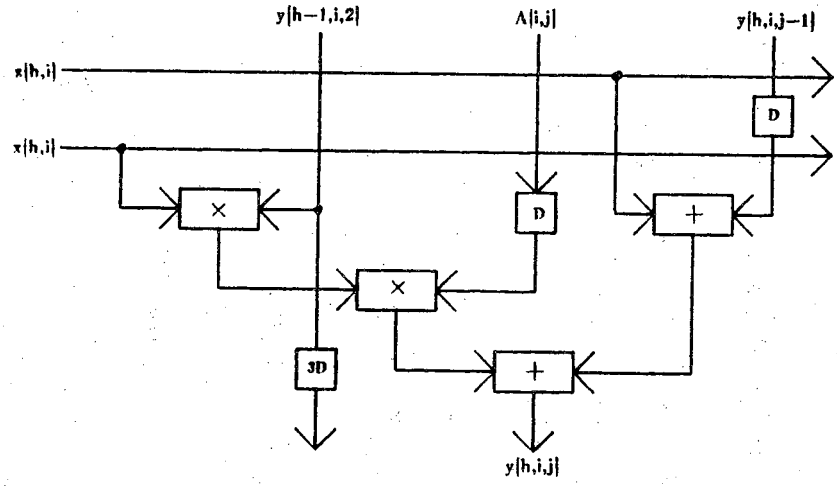


Fig. 4.6: Processing Element for a Type 6c Task

```
        C[j] = C[j−1]
        x[i] = x[i−1]
        y[i,j] = C[j] · (y[i−1,j] * x[i+1] − y[i−1,j+1] * x[i])
                + C[j] · (y[i−1,2] * x[0] − y[i−1,0] * x[2])
    end j
end i
```

where C[j] is a control signal entering the systolic array vertically. The data dependence matrix for the above program is,

$$D = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & -1 & (j-2) & j \end{bmatrix}$$

where the columns from left to right specify data dependence vectors for C[j−1], x[i−1], y[i−1,j], y[i−1,j+1], y[i−1,2] and y[i−1,0] respectively.

The linear indexing matrices for generated variables y[i−1,2] and y[i−1,0] are,

$$I_{By[i-1,2]} = I_{By[i-1,0]} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

The rank of the above linear indexing matrix is less than the dimensionality of the algorithm index set. The variables y[i−1,2] and y[i−1,0] are, therefore, amenable to broadcasting within the systolic array.

By Step 4, the matrix of interconnection primitives, P, is

$$P = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & -1 \end{bmatrix}.$$

Conditions on the selection of the S matrix to avoid broadcasting of specified variables include,

$$S_{By[i-1,2]}\bar{d}_{By[i-1,2]} = C_1 + jC_2$$

and,

$$S_{By[i-1,0]}\bar{d}_{By[i-1,0]} = C_3 + jC_4$$

where C_1, $C_3 \in Z^n$; C_2, $C_4 \in I^n$ and C_2, $C_4 \neq 0$. An S-matrix is selected which satisfies these conditions, minimizes the number of bands, and allows the diophantine equation SD = PK to be solvable for S,

$$S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

for,

$$K = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Now that the S matrix has been chosen, conditions on the selection of the Π transformation may be summarized as follows:

(i) $\Pi\bar{d}_{By[i-1,2]} = C_1 + jC_2$;

(ii) $\Pi\bar{d}_{By[i-1,0]} = C_3 + jC_4$, where $C_1 \geq 2$, $C_3 \geq 0$ and $C_2, C_4 \geq 1$;

(iii) $\Pi\bar{d}_C, \Pi\bar{d}_x \geq 1$;

(iv) $\Pi\bar{d}_{y[i-1,j]}, \Pi\bar{d}_{y[i-1,j+1]} \geq 3$.

The above conditions are derived using Steps 3.1-3.3 of SADLRE. A Π transformation is selected to satisfy these conditions in addition to minimizing the algorithm execution time t, $\Pi = [4\ \ 1]$.

The new transformation matrix, Δ, is found be,

$$\Delta = TD = \begin{bmatrix} 1 & 4 & 4 & 3 & (2+j) & (4+j) \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & -1 & (j-2) & j \end{bmatrix}$$

Information regarding communication latency for the directional flow of variables in the systolic array may be deduced from the Δ matrix. This can be done in a manner similar to that previously described.

The {GVT, UVT, IT} for the recursive vector cross-product operation is shown in Tables 4.10-4.12. The resulting systolic array for this type of operation is delineated in Fig. 4.7(a), and the architecture of an individual processing cell is illustrated in Fig. 4.7(c). The control signal, C(j), operates as an input selection line for the multiplexers. Typically, it will allow y(i−1,j) and y(i−1,j+1) to pass through the multiplexers for the first two rows of the systolic array in Fig. 4.7(a). C(j) will, however, permit y(i−1,0) and y(i−1,j) to propagate through the multiplexers due to the operational specification of a vector cross-product operation. The partitioned architecture for a Type 7 operation is delineated in Fig. 4.7(b). The operational delay for the proposed architecture is 3 time units for $i_{max} = 1$ in the algorithm index set,

Table 4.10: GVT of a Type 7 Task

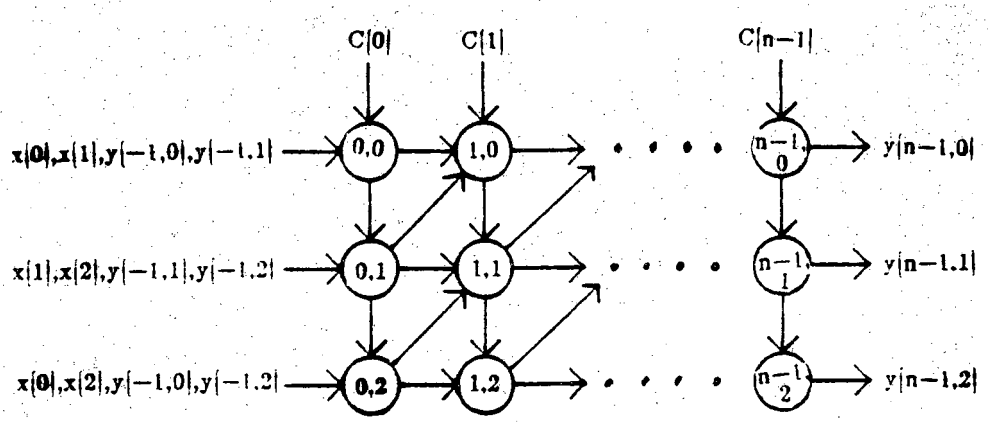
ITERATION (i,j)	BEGIN EXECUTION AT TIME	FINISH EXECUTION AT TIME	IN CELL (i,j)	GENERATING VARIABLE
0,0	0	2	0,0	y(0,0)
0,1	1	3	0,1	y(0,1)
0,2	2	4	0,2	y(0,2)
1,0	4	6	1,0	y(1,0)
1,1	5	7	1,1	y(1,1)
1,2	6	8	1,2	y(1,2)
2,0	8	10	2,0	y(2,0)
2,1	9	11	2,1	y(2,1)
2,2	10	12	2,2	y(2,2)

Table 4.11: UVT of a Type 7 Task

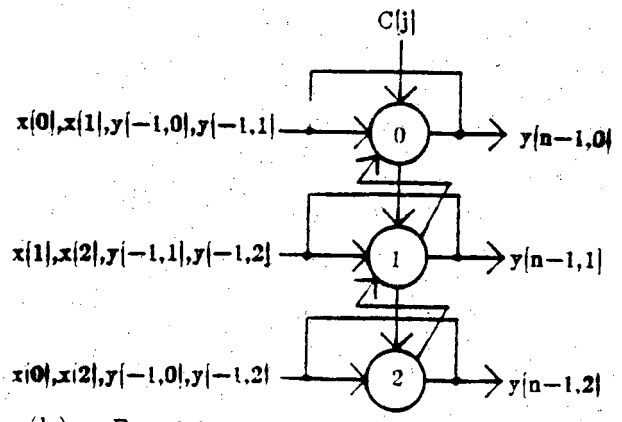
ITERATION (i,j)	USING VARIABLES	FROM CELLS	AT TIMES
0,0	y(-1,0), y(-1,1)	(-1,0), (-1,1)	-2,-1
0,1	y(-1,1), y(-1,2)	(-1,1), (-1,2)	-1,0
0,2	y(-1,0), y(-1,2)	(-1,0), (-1,2)	-2,0
1,0	y(0,0), y(0,1)	(0,0), (0,1)	2,3
1,1	y(0,1), y(0,2)	(0,1), (0,2)	3,4
1,2	y(0,0), y(0,2)	(0,0), (0,2)	2,4
2,0	y(1,0), y(1,1)	(1,0), (1,1)	6,7
2,1	y(1,1), y(1,2)	(1,1), (1,2)	7,8
2,2	y(1,0), y(1,2)	(1,0), (1,2)	6,8

Table 4.12: IT of a Type 7 Task

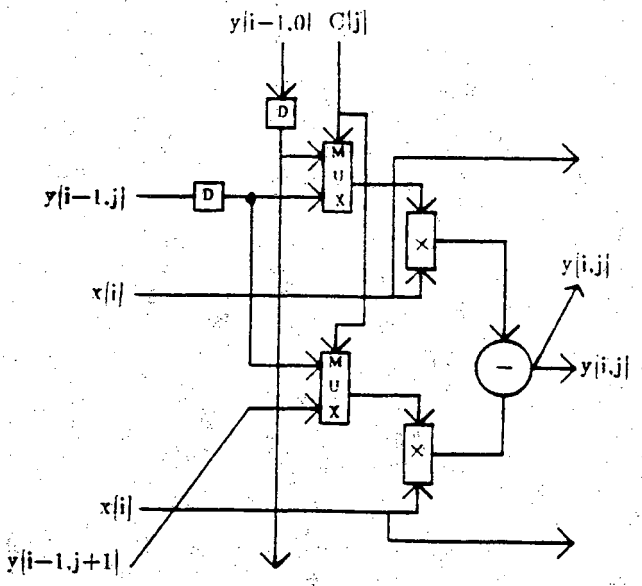
ITERATION (i,j)	USING INPUTS	FROM CELLS	AT TIMES
0,0	x(0), x(1), C(0,0)	(-1,0), (-1,0), (0,-1)	-4,-4,-1
0,1	x(1), x(2), C(0,1)	(-1,1), (-1,1), (0,0)	-3,-3,0
0,2	x(0), x(2), C(0,2)	(-1,2), (-1,2), (0,1)	-2,-2,1
1,0	x(0), x(1), C(1,0)	(0,0), (0,0), (1,-1)	0,0,3
1,1	x(1), x(2), C(1,1)	(0,1), (0,1), (1,0)	1,1,4
1,2	x(0), x(2), C(1,2)	(0,2), (0,2), (1,1)	2,2,5
2,0	x(0), x(1), C(2,0)	(1,0), (1,0), (2,-1)	4,4,7
2,1	x(1), x(2), C(2,1)	(1,1), (1,1), (2,0)	5,5,8
2,2	x(0), x(2), C(2,2)	(1,2), (1,2), (2,1)	6,6,9



(a): Systolic Architecture



(b): Partitioned Systolic Architecture



(c): An Individual Processing Element

Fig. 4.7: Systolic Architecture for a Type 7 Task

which is also the maximum outermost index limit specification for a recursive vector cross-product operation in the inverse dynamics computational problem. Note that an unit latency for input data synchronization is also included in the operational delay.

4.6 Optimal Buffer Assignment for the Formulation of a Balanced Systolic Architecture to Evaluate the Inverse Dynamics Computational Problem

The computational flow of tasks in the forward iterations of **algorithm NEWTON_EULER** (Section 4.4) as a directed task graph is shown in Fig. 4.8. Each node in this diagram represents a systolic processor of Section 4.5. The nomenclature, t_y *i*, next to each node denotes the type of systolic cell, where the variable *i* identifies the type of cell. Also, the variable *x* in the field T_x inside each node specifies the actual task number.

The bottleneck in achieving maximum throughput in the architecture of Fig. 4.8 is due to the different arrival time of data into the multi-input systolic processing cells. Since computations may not be initiated in cells until all input data are available simultaneously, the pipelined execution time is therefore naturally lengthened. We must therefore balance the task graph to achieve maximum pipelining. This problem may be solved by assigning appropriate amounts of delay (using buffer stages) along some paths of the graph.

The problem of balancing a directed graph by inserting buffers along appropriate paths has been solved by the cut-set theorem [21,22], local correctness criterion [22], and the graph-theoretic approach [5,7]. However, we shall use a more systematic approach by formulating it as an integer linear programming problem [47].

Before formally presenting the procedure which may be used to generate a set of integer linear constraint equations for the balancing problem, the following notation/definitions must be introduced.

Definition 1: A Weighted Graph, WG, corresponds to a directed task graph using a given set of processing units which provides a weight variable ω_i, $i \in Z^n$, along the output edges of every node n_i. The weight, ω_i, constitutes the computational delay within node n_i.

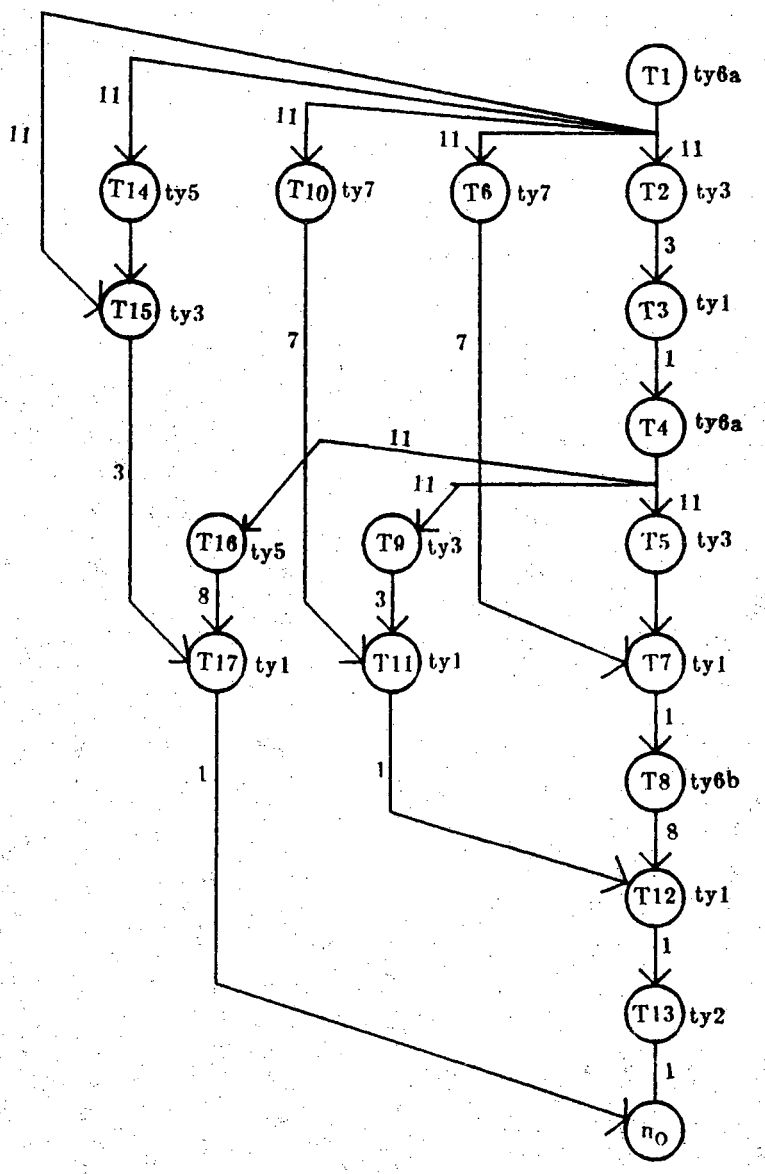


Fig. 4.8: WG for Forward Iterations of N-E Algorithm

Definition 2: A weight, $\sum_{P(n_i)\in P_k(n_i)} \omega(P(n_i))$, is defined as the sum of weights, or total cost, along the kth path, p_k, from the input node to multi-input node, n_i, in the Weighted Graph, WG.

Definition 3: The critical path, $p_c(n_i)$, denotes the path from the input node to multi-input node, n_i which constitutes the largest weight variable, i.e. $\max \sum_{p(n_i)\in p_k(n_i)} \omega(p(n_i))$, in the Weighted Graph, WG.

Definition 4: The Buffered Graph, BG, corresponds to a directed task graph which provides a set of buffers $\{B_j, \ldots, B_{j+n}\}$, $j\in Z^n$, along the (n−j) output edges of multi-output node n_i, $i\in Z^n$, in WG that do not belong to output edges in the critical path $p_c(n_o)$, where n_o is the output node of WG.

Note that an input node, n_I, that supplies operand data for nodes in the directed task graph must be included in BG.

Definition 5: The variable, $\sum_{p(n_i)\in p_k(n_i)} |B(p(n_i))|$, is defined as the sum of buffer variables, or total buffer cost, along the kth path, p_k, from input node, n_I, to multi-input node, n_i, in the Buffered Graph, BG. Now that the nomenclature to be used has been discussed, we may proceed further by presenting a procedure which will generate an optimal buffer assignment for a given weighted graph.

Procedure OBAP (Optimal Buffer Assignment Problem)

Input: Weighted Graph, WG.

Output: Optimal numerical buffer assignment of buffers $B_j, j\in Z^n$, which minimizes the total number of buffers stages in Buffered Graph, BG.

STEP 1: Find critical path, $p_c(n_o)$, to output node n_o from the input node of WG.

STEP 2: Obtain BG by assigning buffer stage variables to the output edges of multi-output nodes in WG, except for the output edges which belong to the critical path, $p_c(n_o)$, of the output node n_o.

STEP 3: For every multi-input node, n_i, except output node n_o, acquire a set of m−1 integer linear constraint equations, where m is the number of inputs into node n_i. These equations may be found by,

$$\sum_{p(n_i)\in p_k(n_i)/p_c(n_i)} [\,|B(p(n_i))| + \omega(p(n_i))\,] = \sum_{p(n_i)\in p_c(n_i)} [\,|B(p(n_i))| + \omega(p(n_i))\,] \quad (4.8)$$

STEP 4: Apply integer linear programming [47] to minimize total number of buffer stages, B_i,

$$\text{Minimize} \sum_{i=1}^{n} |B_i|$$

subject to the constraints of STEP 3, where n is the number of buffers.

The above procedure may now be applied to the weighted graph, WG, for the forward iterations of the Newton-Euler algorithm (in Fig. 4.8). The weights assigned to the output edges of each systolic node represents the computational delay within the node. These delays for the various types of systolic processors developed in Section 4.5 is summarized in Table 4.13.

STEP 1: The critical path $p_c(n_o)$ for the WG of Fig. 4.8 is along the nodes: T1-T2-T3-T4-T5-T7-T8-T12-T13-n_o.

STEP 2: By applying buffer assignment rules to the WG of Fig. 4.8, we obtain the normalized Buffered Graph BG as in Fig. 4.9.

STEP 3: The integer linear constraint equations on the buffering variables in BG may be found by applying Eq. (4.8) to each node:

node T1: $|B6| = 0$
node T2: $|B4| = 11$
node T3: $|B5| = 14$

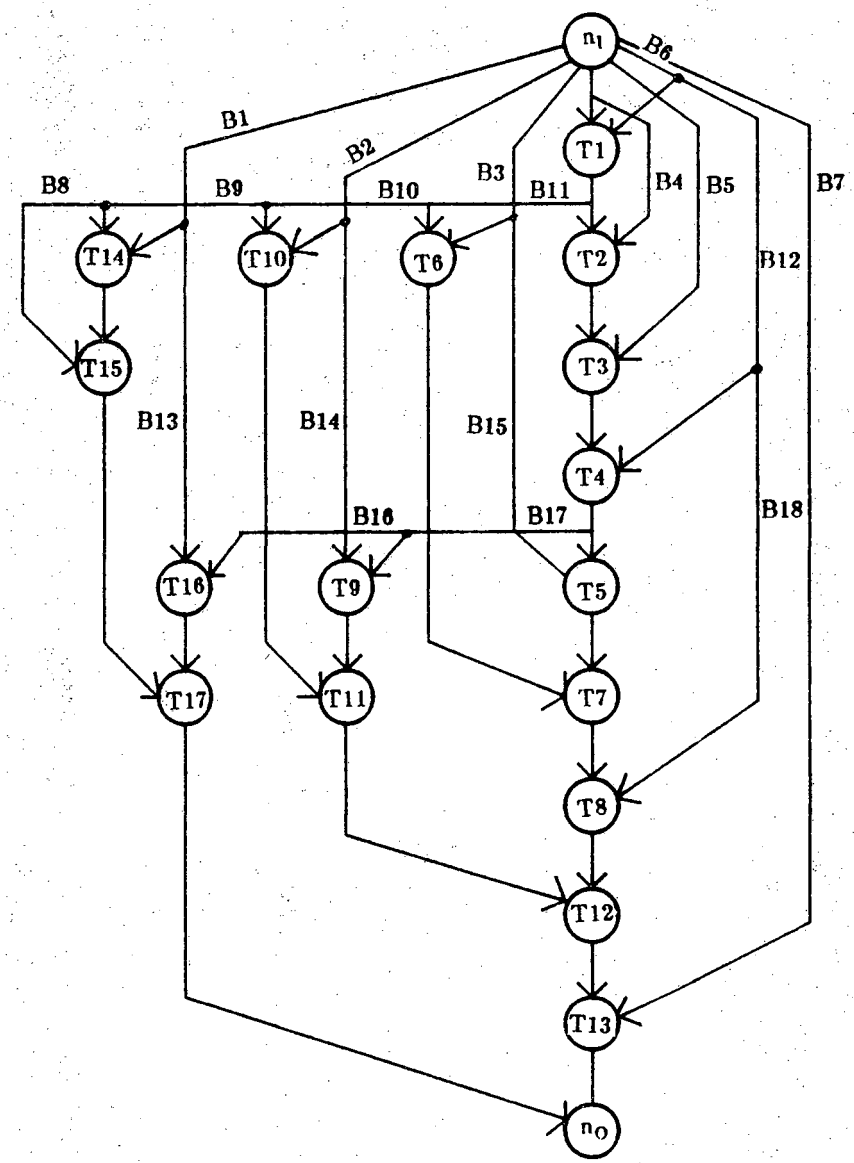


Fig. 4.9: BG for Forward Iterations of N-E Algorithm

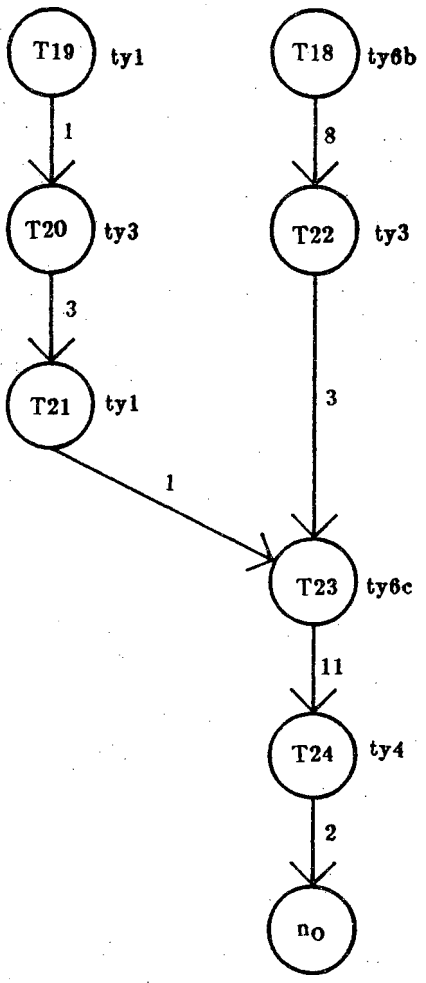


Fig. 4.10: WG for Backward Iterations of N-E Algorithm

node T4: $|B6| + |B12| = 15$
node T5: $|B3| + |B15| = 26$
node T6: $|B3| = |B11| + 11$
node T7: $|B11| = 11$
node T8: $|B6| + |B12| + |B18| = 30$
node T9: $|B2| + |B14| = |B17| + 26$
node T10: $|B2| = |B10| + |B11| + 11$
node T11: $|B10| + |B11| = |B17| + 11$
node T12: $|B17| = 8$
node T13: $|B7| = 39$
node T14: $|B1| = |B9| + |B10| + |B11| + 11$
node T15: $|B8| = 8$
node T16: $|B1| + |B13| = |B16| + |B17| + 26$
node T17: $|B9| + |B10| + |B11| = |B16| + |B17| + 12$

STEP 4: Apply integer linear programming to minimize $\sum_{i=1}^{17} |B_i|$ subject to the constraints of STEP 3. The optimization process generates:

$|B1| = 31, |B2| = 30, |B3| = 22, |B4| = 11,$
$|B5| = 14, |B6| = 0, |B7| = 39, |B8| = 8,$
$|B9| = 1, |B10| = 8, |B11| = 11, |B12| = 15,$
$|B13| = 3, |B14| = 4, |B15| = 4, |B16| = 0,$
$|B17| = 8, |B18| = 15$

Procedure OBAP may now be applied to the WG (in Fig. 4.10) for the backward iterations in program NEWTON-EULER. Intermediate results of steps in the procedure are as follows:

STEP 1: The critical path $p_c(n_o)$ for the WG in Fig. 4.10 is along the nodes: T19-T20-T21-T23-T24-n_o.

STEP 2: Applying buffer assignment rules to the WG of Fig. 4.10, the normalized buffered graph in Fig. 4.11 may be obtained.

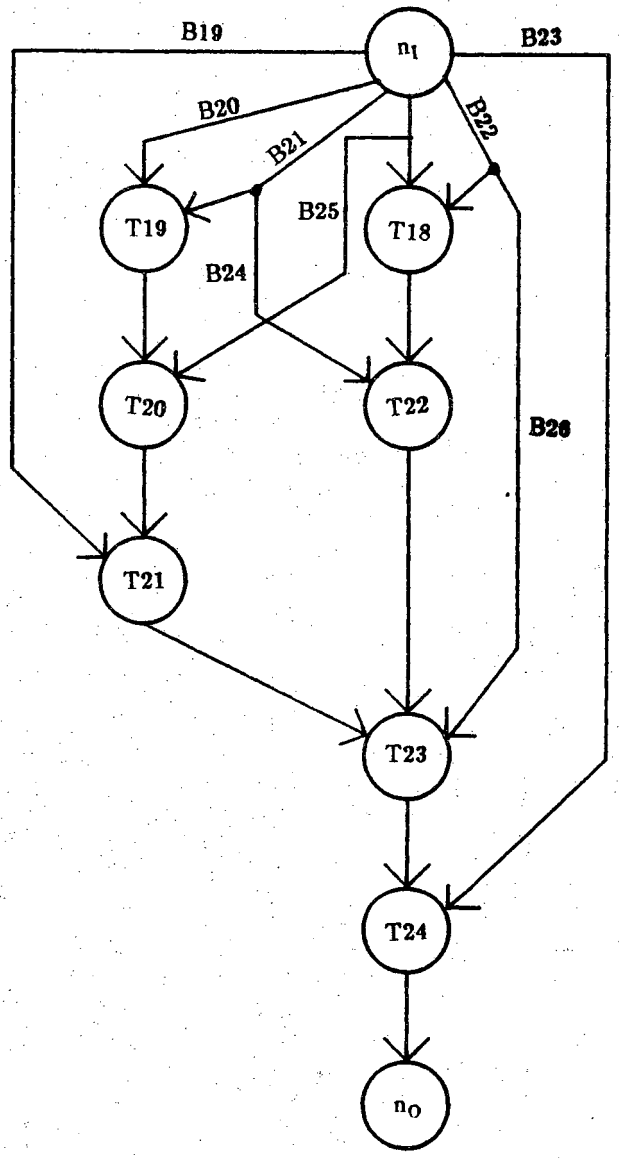


Fig. 4.11: BG for Backward Iterations of N-E Algorithm

Table 4.13: Computational Delays for Basic Set of Systolic Processors

Task Type	Systolic Operational Delay	Systolic Execution Time
1	1	3(n+1)
2	1	3(n+1)
3	3	4(n+1)
4	8	8(n+1)
5	8	8n+10
6a	11	11n+13
6b	8	8n+10
6c	11	11n+13
7	3	4(n+1)

STEP 3: The integer linear constraint equation on buffering variables in Fig. 4.11 are as follows:

node T18: $|B22| = 0$
node T19: $|B20| = |B21|$
node T20: $|B20| + 1 = B25$
node T21: $|B19| = |B20| + 4$
node T22: $|B21| + |B24| = 8$
node T23: $|B20| = 6, |B22| + |B26| = 11$
node T24: $|B23| = 22$

STEP 4: Applying integer linear programming to minimize $\sum_{i=19}^{23} B_i$ subject to the constraints of STEP 3, we get:

$|B19| = 10, |B20| = 6, |B21| = 6, |B22| = 0,$
$|B23| = 22, |B24| = 2, |B25| = 7, |B26| = 11.$

A balanced architecture for the computation of the inverse dynamics problem is thus found by pipelining the weighted systolic architecture of Fig. 4.8 to that of Fig. 4.10 through an intermediate LIFO (Last-In-First-Out) intermediate register file structure. The total time to compute the given problem on the developed systolic machine is $(t_{crf} + t_{crb})n+2$, where t_{crf} is the operational delay of the critical path of Fig. 4.8 (forward iterations), t_{crb} is the operational delay of the critical path in Fig. 4.10 (backward iterations), and n is the number of manipulator joints. By using the operational latency data (Table 4.13) for all types of systolic processors designed in this chapter, the total execution time to evaluate the inverse dynamics problem of an n-link manipulator is 70n+2. Note that the coefficient d_1 (see Theorem 4.1) for our particular systolic machine for the computation of the desired problem is thus 70.

4.7 Conclusions

A novel systolic architecture to compute the inverse dynamics computational problem within the systolic execution time lower bound of $o\lceil d_1 n \rceil$, where d_1 is a specified constant, is discussed. A modified form of the systolic design methodology of Moldovan and Fortes [40] is used as the primary design tool for the architecture. Modifications were required to provide allowances for unequal execution times of processing cells, and to eliminate broadcasting of any input or generated variable

within the systolic array. The inverse dynamics problem is decomposed into a set of unidirectional and linear recurrence tasks amenable for direct mapping onto a fixed systolic system. A set of seven basic types of systolic processor architectures are developed to represent nodes in an acyclic systolic directed task graph to compute the Newton-Euler dynamics algorithm. The balancing of this acyclic task graph is reduced to an integer linear optimization problem for the assignment of delay buffers along alternate data routes. The final systolic architecture achieves maximal pipelining with a minimal number of delay buffers assigned along its various computational paths.

CHAPTER 5
SUMMARY OF CONCLUSIONS

In chapter 2, the ideal lower bounds on the number of parallel processors and execution time to evaluate the inverse dynamics problem of an n-link manipulator in a parallel processing architecture using an optimal scheduling algorithm was proven to be n and $o(\lceil a_1 n \rceil)$ respectively, where a_1 is a specified constant. Next, a novel SIMD scheduling algorithm to perform parallel processing on a SIMD multiprocessor-based architectural model with a crossbar interprocessor network which performs close to the ideal lower bound is presented.. The performance of the N-E dynamics algorithm on the proposed architecture using the specified SIMD scheduling technique is next evaluated. Speedup factor of greater than 3.4 over previous related work [19] for the computation of the defined problem on a parallel processing system is achieved. The SIMD architectural model is then not used again to simulate the performance of PUMA forward and inverse kinematics algorithms because of the non-recursive and non-linear nature of these algorithms. A multiprocessor model with a shared memory interprocessor communication strategy is instead investigated for this purpose using the DFIHS scheduling algorithm. Simulation results show speedup of approximately 2 over the uniprocessor solution when the number of parallel processors is greater than eight and five for the cases of forward and inverse kinematics respectively. Simulation results showed very large speedups is not achieved in this latter multiprocessor architectural model due to high interprocessor communication latency costs. The important results of this chapter may be summarized as in Table 5.1. Note that the hardware overhead due to the crossbar interprocessor network must be included when evaluating the overall system cost for the computation of the inverse dynamics problem.

In chapter 3, we described a cost-efficient parallel and pipelined bit-serial system for the inverse dynamics computational problem to achieve the bit-serial execution time lower bound of $o(\lfloor c_1 k + c_2 kn \rfloor)$. A high performance multi-functional bit-serial

Table 5.1 Execution Times and Hardware Overhead for Computation of Robot Control Algorithms in a MC 68020-Based Multiprocessor System

Robot Control Algorithm	Minimum Execution Time (in μs)	Number of MC 68020's
Inverse Dynamics	513.84	8
PUMA Forward Kinematics	392.60	8
PUMA Inverse Kinematics	750.45	5

Table 5.2 Execution Times and Hardware Overhead for Computation of Inverse Dynamics Problem

Computational Approach	Minimum Execution Time	Hardware Overhead
Parallel Processing	119n	8 PEs, Crossbar Interprocessor Network
Bit-Serial	19k+6kn	32 Bit-Serial Cells
Systolic	70n+2	24 Systolic Processors

cell architecture is described as a building block for the array configurations of the system. Zipper CMOS circuit design strategy is proposed for the cell's implementation to minimize propagation delays and maximize operating frequencies. For the case of a 16-bit system word length and 20MHz operating frequency, the total time to compute the inverse dynamics problem of a 6-link manipulator on the proposed bit-serial architecture is only 44μs.

Finally, in chapter 4, a novel systolic architecture to compute the inverse dynamics computational problem within the systolic execution time lower bound of $o\lceil d_1 n \rceil$, where d_1 is a specified constant, is discussed. A modified form of the systolic design methodology of Moldovan and Fortes [40] is used as the primary design tool for the architecture. Modifications were required to provide allowances for unequal execution times of processing cells, and to eliminate broadcasting of any input or generated variable within the systolic array. The inverse dynamics problem is decomposed into a set of unidirectional and linear recurrence tasks amenable for direct mapping onto a fixed systolic system. A set of seven basic types of systolic processor architectures are developed to represent nodes in an acyclic systolic directed task graph to compute the Newton-Euler dynamics algorithm. The balancing of this acyclic task graph is reduced to an integer linear optimization problem for the assignment of delay buffers along alternate data routes. The final systolic architecture achieves maximal pipelining with a minimal number of delay buffers assigned along its various computational paths.

The performance and hardware complexity of alternate computational structures for the evaluation of the inverse dynamics problem is compared in table 5.1 The variables "n" and "k" in this table specify the manipulator link count and bit-serial system word length respectively. Clearly, the systolic approach achieves superior performance (assuming the word length for the bit-serial system is greater than 16). The hardware complexity for this approach is however highest due to the overhead associated with the control of the systolic arrays and the number of cells in the arrays themselves. The parallel processing approach achieves the next best execution time, where the crossbar network is the primary hardware overhead. Finally, the specified bit-serial architecture provides acceptable performance at the cost of moderate hardware. Lowest amount of hardware complexity is required for this case because of minimum circuitry needed for the implementation of bit-serial cells.

LIST OF REFERENCES

[1] Acosta, R. and Kjelstrup, H.C., "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Trans. Comp.,* Vol. C-35, No. 9, Sept. 1986, pp. 815-828.

[2] Ahmad, S., Meyer, D., and Rahman, M. "On the Design of Special-Purpose Computational Structures for Robot Control: Design Constraints," Proc. 1986 Applied Motion Control Conf., Minneapolis, Minnesota.

[3] Ahmed, H. M., Delosme, J. M., and Morf, M., "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," *Proc. IEEE,* Jan. 1982, pp. 65-82.

[4] Bejczy, A., "Robot Arm Dynamics and Control," Technical Memo 33-669, Jet Propulsion Lab., Pasadena, Calif., 1974.

[5] Brock, J. D., "Translation and Optimization of Dataflow Programs," *Proc. 1979 Int. Conf. Parallel Proc.* , Aug. 1979, pp. 46-54.

[6] Buric, M. and Carver, M., "Bit-Serial Inner Product Processors in VLSI," *CALTECH Int. Conf. on VLSI,* January 1981, pp. 155-164.

[7] Dennis, J. B., and Gao, R. G., "Maximum Pipelining of Array Operations on Static Dataflow Machine," *Proc. 1983 Int. Conf. Parallel Proc.* , Aug. 1983, pp. 331-334.

[8] Denyer, P. and Renshaw, D., *VLSI Signal Processing: A Bit-Serial Approach,* Addison-Wesley, Reading, Mass., 1985.

[9] Du, H. C., "On the Performance of Synchronous Multiprocessors," *IEEE Trans. Comp.,* Vol. C-34, No.5, May 1985, pp. 462-466.

[10] Fernandez, E. B. and Bussel, B., "Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules," *IEEE Trans. Comp.,* Vol. C-22, No. 8, Aug. 1973, pp. 745-751.

[11] Fortes, J. A. B. and Moldovan, D. I., "Parallelism Detection and Algorithm Transformation Techniques useful for VLSI Architecture Design," *J. Parallel Distrib. Comp., May 1985.*

[12] Fortes, J. A. B., and Moldovan, D. I., "Data Broadcasting in Linearly Scheduled Array Processors," *Proc. 11th Int. Symp. on Comp. Arch.* , June 1984, pp. 84-98.

[13] Fu, K. S., Gonzalelez, R. C., Lee, C. S. G., *Robotics: Control, Sensing, Vision, and Intelligence,* Mc Graw Hill, 1987.

[14] Goncalves, N.F. and De Man, H.J., "NORA: A Racefree Dynamic CMOS Technique for Pipelined Logic Structures," *IEEE J. Solid-State Circuits,* Vol. SC-18, No. 3, June 1983, pp. 261-266.

[15] Horowitz, E. and Sahni, S., *Fundamentals of Computer Algorithms,* Computer Science Press, Inc., Rockville, Maryland, 1978.

[16] Hwang, K. and Briggs, F. A., *Computer Architecture and Parallel Processing,* Mc Graw Hill, 1984.

[17] Jensen, E., "The Honeywell experimental distributed processor—An overview," *Computer* , Vol. 11, pp. 28-38, Jan. 1978.

[18] Hollerbach, J., "A Recursive Lagrange Formulation of Manipulator Dynamics and a Comparative Study of Dynamics Formulation Complexity," *IEEE Trans. Sys., Man and Cyber.,* Vol. SMC-10, No. 11, November 1980, pp. 730-736.

[19] Kasahara, H. and Narita, S., "Parallel Processing of Robot-Arm Control Computation on a Multimicroprocessor System," *IEEE Robotics Automat.,* Vol. RA-1, No. 2, June 1985, pp. 104-113.

[20] Krambeck, R.H., Lee, C.M., and Law, H.F.S., "High-Speed Compact Circuits with CMOS," *IEEE J. Solid-State Circuits,* Vol. SC-17, No. 3, June 1982, pp. 614-619.

[21] Kung, S. Y., "VLSI Array Processors," *IEEE ASSP,* July 1985, pp. 4-22.

[22] Kung, S. Y., "Wafer-Scale Integration and Two-Level Pipelined Implementation of Systolic Arrays," *J. Parallel Distrib. Comp.,* Vol. 1, No. 1, Sept. 1984, pp. 36-63.

[23] Kung, S. Y. et. al., "Wavefront Array Processor: Language, Architecture and Applications," *IEEE Trans. Comp.,* Vol. C-31, No. 11, Nov. 1982.

[24] Kung, H. T., "Why Systolic Arrays?," *Proc. IEEE* , January 1982, pp. 37-46.

[25] Lang, T. et. al., "Bandwidth of Crossbar and Multiple Bus Connections for Multiprocessors," *IEEE Trans. Comp.,* Vol. C-31, No. 12, Dec. 1982.

[26] Lathrop, L., "Parallelism in Manipulator Dynamics," M.I.T. Artificial Intelligence Lab., Cambridge, Mass., Tech. Rep. No. 754, Dec. 1983.

[27] Lawler et. al., *The Traveling Salesman,* John Wiley & Sons, Great Britain, 1985.

[28] Lee, C.M. and Szeto, E.W., "Zipper CMOS," *IEEE Circuits and Devices,* May 1986, pp. 10-16.

[29] Lee, C.S.G. and Chang, P.R., "Efficient Parallel Algorithm for Robot Inverse Dynamics Computation," *IEEE Trans. Sys., Man and Cyber.,* Vol. SMC-16, No.4, July/August 1986, pp. 532-542.

[30] Lee, C.S.G. et. al., "Development of the Generalized d'Alembert Equations of Motion for Mechanical Manipulators," *Proc. 1982 Pattern Recognition and Image Processing Conf.,* Las Vegas, Nev., 1982, pp. 634-640.

[31] Li, G. J. and Wah, B. W., "The Design of Optimal Systolic Arrays," *IEEE Trans. Comp.* , Vol. C-34, No. 1, Jan. 1885, pp. 66-77.

[32] Liao, F.Y. and Chern, M.Y., "Robot Manipulator Dynamics Computation on a VLSI Array Processor," *Proc. 1985 Int. Conf. on Supercomputers,* 1985, pp. 116-121.

[33] Lint, B. and Agerwala, T., "Communication Issues in the Design and Analysis of Parallel Algorithms", *IEEE Trans. Soft. Eng.,* Vol. SE-7, No. 2, March 1981, pp. 174-188.

[34] Liu, C.H. and Chen, Y.M., "Multi-Microprocessor Based Cartesian-Space Control Techniques for a Mechanical Manipulator," *IEEE J. Robotics Automat.,* Vol. RA-2, No. 2, June 1986, pp. 110-115.

[35] Luh, J.Y.S. and Lin, C. S., "Scheduling of Parallel Computation for a Computer-Controlled Mechanical Manipulator," *IEEE Trans. Sys., Man and Cyber.,* Vol. SMC-12, No. 2, March/April 1982, pp. 214-234.

[36] Luh, J.Y.S., Walker, M.W. and Paul, R.P.C., "On-Line Manipulator Scheme for Mechanical Robots," *IEEE Trans. Automatic Control,* Vol. AC-25, No. 3, pp. 468-474.

[37] Lyon, R.F., "Two's Complement Pipeline Multipliers," *IEEE Trans. Commun.,* Apr. 1976, pp. 418-425.

[38] Masson, G. M. et. al., "A Sampler of Circuit Switching Networks", *Proc. IEEE* , June 1979, pp. 145-160.

[39] Mead, C. and Conway, L., *Introduction to VLSI Systems,* Addison-Wesley, Reading, Mass., 1980.

[40] Moldovan, D. I. and Fortes, J. A. B., "Partitioning and Mapping Algorithms into Fixed Systolic Arrays," *IEEE Trans. Comp.* , Vol. C-35, No. 1, Jan. 1986, pp. 1-12.

[41] Moldovan, D.I., "On the Analysis and Synthesis of VLSI Algorithms," *IEEE Trans. Comp.,* Vol. C-31, No. 11, Nov. 1982, pp. 1121-1126.

[42] Murty, K. G., *Linear Programming,* John Wiley & sons, Inc.,1983.

[43] Nigam, R. and Lee, C.S.G., "A Multiprocessor-Based Controller for the Control of Mechanical Manipulators," *IEEE J. Robotics Automat.,* Vol. RA-1, No. 4, Dec. 1985, pp. 173-182.

[44] Ohwada, N. et. al., "LSI's for Digital Signal Processing," *IEEE J. Solid-State Circuits,* Vol. SC-14, No. 2, Apr. 1979, pp. 214-220.

[45] Orin, D.E., Chao, H.H. and Schrader, W.W., "Pipeline/Parallel Algorithms for the Jacobian and Inverse Dynamics Computations," *Proc. 1985 IEEE Int. Conf. Robotics,* pp. 785-789.

[46] Orin, D.E., "Kinematic and Kinetic Analysis of Open-Chain Linkages Utilizing Newton-Euler Methods," *Math. Biosc.* , Vol. 43, 1979, pp. 107-130.

[47] Papadimitrious, C. H. and Steiglitz, K., "Combinatorial Optimization: Algorithms and Complexity," Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[48] Quinton, P., "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations," *IEEE Conf. Par. Proc.* , 1984, pp. 208-214.

[49] Rahman, M. and Meyer, D., "Case Study—A MC 68020 Based I/O Controller", in *Topics in High-Level Computer Architecture* , Milutinovic, V. ed., Computer Science Press, Rockville, Maryland, to appear, Aug. 1987.

[50] Rahman, M. and Meyer, D., "High-Performance Parallel and Pipelined Bit-Serial Architecture for Robot Inverse Dynamics Computations," Submitted to *IEEE Trans. Sys., Man, Cyber.,* January 1987.

[51] Siegel, L. et. al., "Performance Measures for Evaluating Algorithms for SIMD Machines," *IEEE Trans. Soft. Eng.,* Vol. SE-8, No. 4, July 1982, pp. 319-330.

[52] Siegel, H. J., "Interconnection Networks for SIMD Machines," *Proc. IEEE* , June 1979, pp. 57-65.

[53] Siegel, H. J., "A Model of SIMD Machines and a Comparison of Various Interconnection Networks," *IEEE Trans. Comp.* , Vol. C-28, No.12, Dec. 1979, pp. 907-917.

[54] Siegel, H. J., "The Theory Underlying the Partitioning of Permutation Networks," *IEEE Trans. Comp.* , Vol. C-29, No. 9, Sept. 1980, pp. 791-800.

[55] Tjaden, G. S. and Flynn, M. J., "Detection and Parallel Execution of Independent Instructions," *IEEE Trans. Comp.* , Vol. C-19, No. 10, Oct. 1970.

[56] Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J.* , Jan. 1967, pp. 25-33.

[57] Walker, M.W. and Orin, D.E., "Efficient Dynamics Computer Simulation of Robotic Mechanisms," *Robot Motion,* Brady, M., ed., MIT Press, 1982, pp. 107-125.

[58] Watanabe, T. et. al., "Improvement in the Computing Time of Robot Manipulators Using a Multimicroprocessor," *IEEE Trans. ASME,* Vol. 108, Sept. 1986, pp. 190-197.

[59] Wong, Y. and Delosme, J. M., "Optimal Systematic Implementations of N-Dimensional Recurrences," *Int. Symp. Comp. Arch.* , June 1985, pp. 618-621.

[60] Zheng, Y.F. and Hemami, H., "Computation of Multibody System Dynamics by a Multiprocessor Scheme," *IEEE Trans. Sys., Man, and Cyber.,* SMC-16, No. 1, Jan./Feb. 1986, pp. 102-110.

Appendix A: N-E Equations of Motion for a Manipulator with Rotary Joints

Forward Iterations:

FOR i = 0 TO (n−1) DO

BEGIN

$$^{i+1}\omega_{i+1} = {}_{i}^{i+1}R(^{i}\omega_i + \dot{\theta}_{i+1}^{z})$$

$$^{i+1}\dot{\omega}_{i+1} = {}_{i}^{i+1}R(^{i}\dot{\omega}_i + \ddot{\theta}_{i+1}^{z} + {}^{i}\omega_i \times \dot{\theta}_{i+1}^{z})$$

$$^{i+1}\dot{v}_{i+1} = {}_{i}^{i+1}R\,{}^{i}\dot{v}_i + (^{i+1}\dot{\omega}_{i+1} \times {}^{i}P_{i+1}) + {}^{i+1}\omega_{i+1} \times (^{i+1}\omega_{i+1} \times {}^{i}P_{i+1})$$

$$^{i+1}\dot{v}_{c_{i+1}} = (^{i+1}\dot{\omega}_{i+1} \times {}^{i+1}P_{c_{i+1}}) + {}^{i+1}\omega_{i+1} \times (^{i+1}\omega_{i+1} \times {}^{i+1}P_{c_{i+1}}) + {}^{i+1}\dot{v}_{i+1}$$

$$^{i+1}F_{i+1} = m_{i+1}\,{}^{i+1}\dot{v}_{c_{i+1}}$$

$$^{i+1}N_{i+1} = {}^{i+1}I_{i+1}\,{}^{i+1}\dot{\omega}_{i+1} + {}^{i+1}\omega_{i+1} \times (^{i+1}I_{i+1}\,{}^{i+1}\omega_{i+1})$$

END

Backward Iterations:

FOR i = n TO 1 DO

BEGIN

$$^{i}f_i = {}^{i}F_i + {}_{i+1}^{i}R\,{}^{i+1}f_{i+1}$$

$$^{i}n_i = {}^{i}N_i + (^{i}P_{i+1} + {}^{i}P_{c_{i+1}}) \times {}^{i}F_i + {}_{i+1}^{i}R(^{i+1}n_{i+1} + {}^{i}P_{i+1} \times {}^{i+1}f_{i+1})$$

$$\tau_i = {}_{i-1}^{i}R^{z}\,{}^{i}n_i$$

END

where,

m_i	**mass of link i**
$\dot{\theta}_{i+1}^{z}$	**zth component of the joint velocity of link i+1**
$\ddot{\theta}_{i+1}^{z}$	**zth component of the joint acceleration of link i+1**
$^{i+1}\omega_{i+1}$	**angular velocity of link i+1 with respect to the i+1th coordinate frame**
$^{i+1}\dot{\omega}_{i+1}$	**angular acceleration of link i+1 with repect to the i+1th coordinate frame**
$^{i+1}\dot{v}_{i+1}$	**linear acceleration of link i+1 with respect to the i+1th coordinate frame**
$^{i+1}\dot{v}_{c_{i+1}}$	**linear acceleration of the center of mass of link i+1 with respect to the i+1th fra**

${}_{i}^{i+1}R$	**rotation matrix which maps position vectors from i+1th frame to frame i**
${}^{i}P_{i+1}$	**origin of link i+1 with respect to the ith coordinate frame**
${}^{i+1}P_{c_{i+1}}$	**location of the center of mass of link i+1 with respect the i+1th coordinate fram**
${}^{i+1}F_{i+1}$	**total inertial force exerted on link i+1 at the center of mass**
${}^{i+1}N_{i+1}$	**total moment exerted on link i+1 at the center of mass**
${}^{i}f_{i}$	**force exerted on link i by link i−1**
${}^{i}n_{i}$	**moment exerted on link i by link i−1**
τ_i	**torque exerted by the actuator on link i**

Appendix B: 3-D Matrix/Vector Arithmetic Operations Used In The N-E Dynamics Computational Problem

(1) **Vector-Add (VA)**

The vector sum Y_i of vectors a_i and b_i is defined as,

$$Y_i = (a_i + b_i) \quad \text{for } i = 1,2,3.$$

(2) **Scalar-Vector (SV) product**

The scalar-vector product Y_i of vector a_i by scalar K is defined as,

$$Y_i = Ka_i \quad \text{for } i = 1,2,3.$$

(3) **Inner product (IP)**

The inner product Y of vectors a_i and b_i is defined as,

$$Y = \sum_{i=1}^{3} a_i b_i.$$

(4) **Matrix-vector (MV) product**

The matrix-vector product Y_n of matrix a_{ni} and vector b_i is defined as,

$$Y_n = \sum_{i=1}^{3} a_{ni} b_i \quad \text{for } n = 1,2,3.$$

(5) **Vector cross (VC) product**

The vector cross product, Y of vectors a_i and b_i, is defined as,

$$Y = Y_i + Y_j + Y_k = \det \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

Appendix C: PUMA Forward Kinematics Solution

The position and orientation of the end-effector of the PUMA arm with respect to a fixed reference coordinate system, given the joint angles and and geometric link parameters, may be found as follows:

The arm matrix for the PUMA robot arm is

$$\mathbf{T} = \mathbf{T}_1\,\mathbf{T}_2 = {}^0\mathbf{A}_1\,{}^1\mathbf{A}_2\,{}^2\mathbf{A}_3\,{}^3\mathbf{A}_4\,{}^4\mathbf{A}_5\,{}^5\mathbf{A}_6 = \begin{bmatrix} n_x & s_x & a_x & p_x \\ n_y & s_y & a_y & p_y \\ n_z & s_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where,

$$n_x = C_1[C_{23}(C_4C_5C_6 - S_4S_6) - S_{23}S_5C_6] - S_1(S_4C_5C_6 + C_4S_6)$$

$$n_y = S_1[C_{23}(C_4C_5C_6 - S_4S_6) - S_{23}S_5C_6] + C_1(S_1C_5C_6 + C_4S_6)$$

$$n_z = -S_{23}[C_4C_5C_6 - S_4S_6] - C_{23}S_5C_6$$

$$s_x = C_1[-C_{23}(C_4C_5S_6 + S_4C_6) + S_{23}S_5S_6] - S_1(-S_4C_5S_6 + C_4C_6)$$

$$s_y = S_1[-C_{23}(C_4C_5S_6 + S_4C_6) + S_{23}S_5S_6] + C_1(-S_4C_5S_6 + C_4C_6)$$

$$s_z = S_{23}(C_4C_5S_6 + S_4C_6) + C_{23}S_5S_6$$

$$a_x = C_1(C_{23}C_4S_5 + S_{23}C_5) - S_1S_4S_5$$

$$a_y = S_1(C_{23}C_4S_5 + S_{23}C_5) + C_1S_4S_5$$

$$a_z = -S_{23}C_4S_5 + C_{23}C_5$$

$$p_x = C_1[d_6(C_{23}C_4S_5 + S_{23}C_5) + S_{23}d_4 + a_3C_{23} + a_2C_2] - S_1(d_6S_4S_5 + d_2)$$

$$p_y = S_1[d_6(C_{23}C_4S_5 + S_{23}C_5) + S_{23}d_4 + a_3C_{23} + a_2C_2] + C_1(d_6S_4S_5 + d_2)$$

$$p_z = d_6(C_{23}C_5 - S_{23}C_4S_5) + C_{23}d_4 - a_3S_{23} - a_2S_2$$

Appendix D: PUMA Inverse Kinematics Solution

The inverse kinematics problem can be stated as: Given the position/orientation of the manipulator hand and the link/joint parameters, determine the joint angles so that the manipulator can be positioned as desired. That is, given

$$\mathbf{T} = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{n} & \mathbf{o} & \mathbf{a} & \mathbf{p} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

the joint angle equations are [2]:

$$r = (p_x^2 + p_y^2)^{1/2}$$

$$\theta_1 = \tan^{-1}\left[\frac{p_y}{p_x}\right] - \tan^{-1}\left[\frac{d_3}{\pm\sqrt{r^2 - d_3^2}}\right]$$

$$f_{11p} = p_x C_1 + p_y S_1$$

$$f_{11o} = o_x C_1 + o_y S_1$$

$$f_{13p} = -p_x S_1 + p_y C_1$$

$$f_{13o} = -o_x S_1 + o_y C_1$$

$$f_{11a} = a_x C_1 + a_y S_1$$

$$f_{13a} = -a_x S_1 + a_y C_1$$

$$d = f_{11p}^2 + f_{12p}^2 - d_4^2 - a_3^2 - a_2^2$$

$$e = 4a_2^2 a_3^2 + 4a_2^2 a_4^2$$

$$\theta_3 = \tan^{-1}\left[\frac{a_3}{-d_4}\right] - \tan^{-1}\left[\frac{d}{\pm\sqrt{e - d^2}}\right]$$

$$\theta_{23} = \tan^{-1}\left[\frac{w_2 f_{11p} - w_1 p_z}{w_1 f_{11p} + w_2 p_z}\right]$$

where $w_1 = a_2 C_3 + a_3$, $w_2 = d_4 + a_2 S_3$, $C_i \equiv \cos\theta_i$, and $S_i \equiv \sin\theta_i$.

$$\theta_2 = \theta_{23} - \theta_3$$

$$\theta_4 = \tan^{-1}\left[\frac{-S_1 a_x + C_1 a_y}{C_{23}(C_1 a_x + S_1 a_y) - S_{23} a_z}\right]$$

$$= \tan^{-1}\left[\frac{f_{13a}}{C_{23} f_{11a} - S_{23} a_z}\right]$$

$$\theta_5 = \tan^{-1}\left[\frac{C_4[C_{23}(C_1 a_x + S_1 a_y) - S_{23} a_z] + S_4[-S_1 a_x + C_1 a_y]}{S_{23}(C_1 a_x + S_1 a_y) + C_{23} a_z}\right]$$

$$= \tan^{-1}\left[\frac{C_4(C_{23} f_{11a} - S_{23} a_z) + S_4 f_{13a}}{S_{23} f_{11a} + C_{23} a_z}\right]$$

$$\theta_6 = \tan^{-1}\left[\frac{-C_5[C_4(C_{23} f_{11o} - S_{23} o_z) + S_4 f_{13o}] + S_5(S_{23} f_{11o} + C_{23} o_z)}{-S_4(C_{23} f_{11o} - S_{23} o_z) + C_4 f_{13o}}\right]$$

where $(-\frac{d_4}{a_2})$, $(\frac{d_4}{a_2})$, $(\frac{a_3}{a_2})$ are constants, $C_{ij} \equiv \cos(\theta_i + \theta_j)$, and $S_{ij} \equiv \sin(\theta_i + \theta_j)$.

Appendix E: DFIHS (Depth-First-Implicit-Heuristic-Search) Algorithm

The DF/IHS is a kind of depth first/heuristic search. The main feature of DF/IHS lies in the fact that, unlike the conventional DF/H method, it does not require the computation of the values of heuristic function for all active nodes with the largest depth in order to find, as the next branching node, the node having the smallest value. Prior to the search procedure, priorities are assigned to those nodes that may be generated during search by using the priority list of the CP/MISF method. In this manner, the memory requirements and the average computing time required for search can be reduced markedly, since the choice of the proper next branching node can be made without computing the heuristic values. The DF/IHS method is broken down into two parts, the preprocessing part to assign priorities heuristically to the nodes generated during search, and the depth-first search part.

1) *Preprocessing (Task Renumbering):* All tasks are renumbered in the same order as in the priority list of the CP/MISF method. This can be performed by the use of an $O(n^2)$ algorithm involving the solution of a longest-path problem and sorting.

2) *Depth-First Search:* In order to describe the features of DF/IHS, use is made of Kohler's general representation of the branch-and-bound method, *BB(Bp, S, E, F, D, L, U, ∈, RB)*. (The dominant relation *D* and the characteristic function *F* are not used in DF/IHS).

   a) Branching Rule Bp:

      1) The original problem is decomposed along the time axis (an example of decomposition to subproblems).

      2) Since optimal solutions may not be obtained by simply assigning the ready task(s) to the available processor(s), a fictitious task that forces a processor to be idle is introduced.

      3) A total of

$$n_{branch} = (n_{ready} + n_{idle}) \, C \, m_{av}$$

      nodes are generated from each node in the search tree, where n_{ready} is the number of ready tasks at time *t:* n_{idol} is the number of idle tasks to be considered at time *t* (if $m_{av} = m$ then $n_{idle} =$

m_{av} - 1; if $1 \le m_{idle} = m_{av}$); m_{av} is the number of processors available at time *t;* C means combination.

b) *Selection is made in the form of DF/FIFO.*

1) Selection is made in the form of DF/FIFO.

2) A special table *R* (Ready task table) and pointer *SP* (next branching node selection pointer) for the memory of active nodes and selection of a next branching node are used to facilitate node selection by the computation of the order *O(m).* At the same time, the storage requirement, which often becomes the bottleneck in using the branch-and-bound method, can be retained in the order *O(m·n).*

3) As the initial search solution, the CP/MISF solution obtained has an error bound, relative to the optimal solution, that is guaranteed. Thus any intermediate solution which has been obtained when the search procedure is terminated is superior to the CP/MISF solution in its accuracy.

c) *Lower Bound Function L:*

1) Fernadez' extension of Hu's lower bound

$$t_{hu}(\pi_a) = t_{cr}(\pi_a) + [q(\pi_a)]$$

$$q(\pi_a) = \max_{0 \le t_{hu} \le t_{cr} \le (\pi_a) t_0} \left\{ -t_k + (1/m) \int_0^{t_k} F(\bar{\tau}, t)\, dt \right\}$$

where the load density function $F(\bar{\tau}_j, t)$ is given by

$$\bar{\tau}_j = t_{cr}(\pi_a) - t_j - t_0$$

$$f(\bar{\tau}_j, t) \begin{cases} = 1, & \text{for } t \in (\bar{\tau}, \bar{\tau}_j + t_j) \\ = 0, & \text{otherwise} \end{cases}$$

$$F(\bar{\tau}, t) = \sum_{j \in (\pi_a)} f(\bar{\tau}_j, t)\,,$$

is used in the light of the accuracy and the computing load involved. Here π_a is the partial problem decomposed by branching operations; t_0 is the allocation time represented by the current branching node; and [x] is the minimum integer greater

than x.

2) Since the calculation of the lower-bound function in 1) requires the use of a pseudo-polynomial time algorithm, two much simpler lower bounds with the complexity of at most $O(n)$ are used jointly so that the computing load for the bounding operation is minimized. Specifically, the computing load is much reduced by applying the bounding operations by the lower bound of 1) only to those nodes for which bounding by using the simple bounds has failed.

d) *Upper Bound Cost U:* Since a very accurate intermediate solution, i.e., U, can be obtained as the initial solution, the total search time can be reduced remarkably.

e) *Elimination Rule E:* E = U/DBAS (upper bound tested for dominance of descendants of branching node and members of currently active set).

f) *Acceptable Approximation Error* ϵ*:*

1) While Kohler could only evaluate the relative error of the intermediate solution $\hat{U}$ from the lower bound as "Bracket," $BR((\hat{U} - L)/\hat{U}SBR)$. DF/IHS can obtain an approximate solution whose relative error from the optimal solution does not exceed ϵ.

$$(\hat{U} - t_{opt})/t_{opt} \leq \epsilon \ .$$

2) Useless search computation can be avoided since the search process is terminated as soon as optimality is attained or a predetermined approximation error limit is reached by comparing the lower bound with the intermediate solution $\hat{U}$ or the approximate intermediate solution $\hat{U}_\epsilon$, $\hat{U}_\epsilon = \hat{U}/(1+\epsilon)$.

g) *Resource Bound RB:* In the conventional scheduling algorithms on the basis of the branch-and-bound method, the number of active nodes increases exponentially with the number of tasks n, and because of the limit of memory capacity, only small problems of about 40 tasks could be scheduled. By the use of special lists and pointers, however, DF/IHS can successfully reduce the number of active nodes for each of which it is required to memorize the state. The search process only requires the memory area of the order $O(m \cdot n)$. Thus the resource bound, i.e., storage limit, it rarely violated.